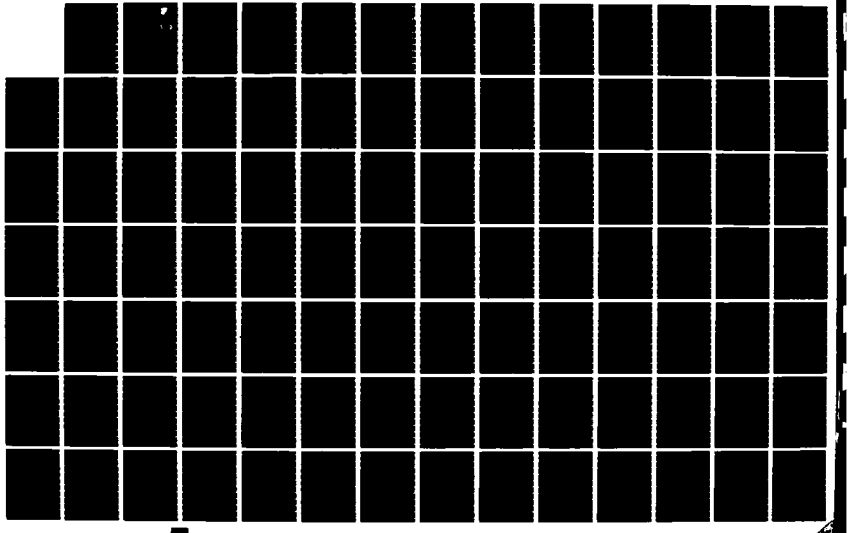
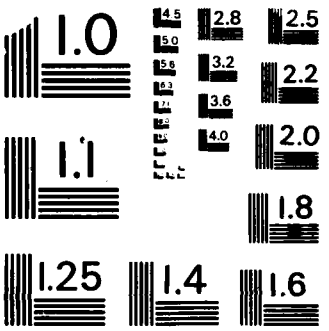
AD-A169 754 DECENTRALIZED SYSTEM CONTROL(U) CARNEGIE-MELLON UNIV 1/4
PITTSBURGH PA DEPT OF COMPUTER SCIENCE
E D JENSEN ET AL. APR 86 CMU-CS-ARCHONS-83-1
UNCLASSIFIED RADC-TR-85-199 F30602-81-C-0297 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

AD-A169 754

RADC-TR-85-199
Final Technical Report
April 1986

DECENTRALIZED SYSTEM CONTROL

DTIC
ELECTE
JUL 25 1986
S D

Carnegie-Mellon University

E. Douglas Jensen, Raymond K. Clark, Robert P. Colwell, Charlie Y. Hitchcock, Chuck P. Kollar, C. Douglass Locke, John P. Lehoczky, J. Duane Northcutt, Norm L. Pleszkoch, Peter M. Schwarz, Lui Sha, Samuel E. Shipman, Hide Tokuda, James W. Wendorf and Roli G. Wendorf

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC FILE COPY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreing nations.

RADC-TR-85-199 has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Command & Control Division

FOR THE COMMANDER: Richard W. Pouliot

RICHARD W. POULIOT
Plans & Programs Division

DESTRUCTION NOTICE - For classified documents, follow the procedures in DOD 5200.22-M, Industrial Security Manual, Section II-19 or DOD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A169 754

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-ARCHONS-83-1	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-85-199
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Dept. of Computer Science Pittsburgh PA 15213	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0297
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62702F; PROJECT NO. 5581; TASK NO. 21; WORK UNIT ACCESSION NO. 47

11. TITLE (Include Security Classification)
DECENTRALIZED SYSTEM CONTROL

12. PERSONAL AUTHOR(S) E. Douglas Jensen, Raymond K. Clark, Robert P. Colwell, Charlie Y. Hitchcock, Chuck P. Kollar, C. Douglass Locke, John P. Lehoczky, J. Duane Northcutt, Norm L. Pleszkoch, (over)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Nov 81 TO Mar 84	April 1986	320

16. SUPPLEMENTARY NOTATION
N/A

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Archons; ArchOS; decentralized control; decentralized algorithm simulator; interprocess communication; Distributed System (over)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes progress on research in decentralized control conducted during the reporting period as part of the Archons Project. The report focuses on fundamental issues of decentralized control ranging from investigations of decentralized resource management principles to architectural support for decentralized operating systems. A development plan for the decentralized ArchOS operating system is included. Also, the DATE decentralized algorithm simulation environment and the Archons Interim Testbed are described.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315) 330-2158 22c. OFFICE SYMBOL RADC (COTD)

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 12. Personal Author(s) (Cont'd)

Peter M. Schwarz, Lui Sha, Samuel E. Shipman, Hide Tokuda, James W. Wendorf, Roli G. Wendorf

Block 18. Subject Terms (Cont'd)

Distributed Operating System

Table of Contents

1. Executive Summary 1

1.1 Archons and ArchOS Objectives 1
1.2 Summary of This Period's Tasks 3
1.2.1 Decentralized Resource Management Principles 3
1.2.2 ArchOS 4
1.2.3 Predominant OS Functions 4
1.2.4 Transactions 5
1.2.5 Interprocess Communication 5
1.2.6 Decentralized Algorithm Testing Environment 5
1.2.7 Decentralized Computer Architecture 6
1.2.8 Interim Testbed 6

2. ArchOS: A Decentralized Operating System 8

2.1 Overview 8
2.2 ArchOS Objectives 8
2.2.1 Background 8
2.2.2 Decentralized Resource Management 10
2.2.2.1 Logically Decentralized Resource Management 11
2.2.2.2 Physically Decentralized Resource Management 12
2.2.3 Other Objectives 17
2.2.3.1 Research Per Se Versus Facility Development 17
2.2.3.2 Large Scale Experimental Computer System Research 18
2.2.3.3 Application and Attributes 19
2.3 ArchOS Development Plan 20
2.3.1 Research Requirements 20
2.3.2 Clients Interface Specification 22
2.3.3 System Functionality Specification 24
2.3.4 System Architectural Specification 24
2.3.5 System Design Specification 25
2.3.6 Component Design Specification 26
2.3.7 Implementation 26
2.4 Conclusion 27
2.5 References 27

3. Transactions for Operating Systems 30

3.1 Overview 30
3.2 Relational Data Model 30
3.2.1 Introduction 30
3.2.2 The Relational Model of Data Consistency 31
3.2.2.1 Our Objections to the Serialization Model 31
3.2.2.2 Classification of Relations 33
3.2.2.3 Definitions 33
3.2.2.4 Representations of Data Objects and Data Invariants 34
3.2.2.5 Some Important Observations 34
3.3 A Modular Approach to Non-serializable Concurrency Control: Database Consistency, Transaction Correctness, and Schedule Optimality 35
3.3.1 Introduction 36

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification .....
By .....
Distribution /
Availability Codes
Dist Avail and / or Special
A-1
DTIC COPY INSPECTED

3.3.2 A Model of Operating System Database 39
3.3.2.1 Consistency Preserving Partition of Database --- An Example 40
3.3.2.2 Data Objects, Database and Consistency 41
3.3.2.3 Consistency Preserving Partition of Database 42
3.3.3 A Model for Transaction Systems 46
3.3.3.1 Single Level Transactions 47
3.3.3.1.1 Syntax 47
3.3.3.1.2 Schedules and Setwise Serializability 49
3.3.3.1.3 Consistency and Correctness 52
3.3.3.1.4 Algorithms for Maintaining Setwise Serializability 56
3.3.3.2 Nested Transactions 57
3.3.3.2.1 Syntax 57
3.3.3.2.2 Consistency and Correctness 58
3.3.3.3 Compound Transactions 59
3.3.3.3.1 Consistency Preserving Partition of Transactions --- An Example 60
3.3.3.3.2 Syntax 63
3.3.3.3.3 Consistency and Correctness 63
3.3.4 Modularity, Application Independence and Optimality 65
3.3.4.1 Modularity and Application Independence 65
3.3.4.2 Optimality and Completeness 69
3.3.5 Conclusion 73
3.4 Distributed Cooperating Processes and Transactions 74
3.4.1 Co-operating Processes 74
3.4.1.1 A New Formulation 74
3.4.1.2 Example: Remote Process Interruption and Abortion 76
3.4.1.3 Example: Process Creation and Destruction 78
3.4.2 Co-operating Transactions 80
3.4.2.1 A New Concept 80
3.4.2.2 Example: Graceful Degradation 82
3.4.2.3 Example: Distributed Load Leveling 83
3.4.3 Conclusion 85
3.5 References 85

4. Interprocess Communication 89
4.1 Overview 89
4.2 The Separation of Policy and Mechanism in IPC 90
4.2.1 Introduction 90
4.2.2 Background 92
4.2.2.1 Definition of Interprocess Communication 92
4.2.2.1.1 The Role of IPC in Programming Systems 93
4.2.2.1.2 The IPC Facility Design Space 93
4.2.2.1.3 Our Scope of Interest in the Universe of IPC Facilities 95
4.2.2.2 The Separation of Policy and Mechanism 95
4.2.2.2.1 Policy/Mechanism Separation as a General Structuring Methodology 95
4.2.2.2.2 The Separation of Policy and Mechanism in IPC 97
4.2.2.3 Interprocess Communication for Decentralized Computer Systems 97
4.2.3 Rationale 98
4.2.3.1 Significance of Interprocess Communication Facility Design and Implementation 99

4.2.3.2 Alternative Approaches to Flexible Interprocess Communication Facilities 100
4.2.3.2.1 A "Parameterized" Approach 100
4.2.3.2.2 A "Strictly Layered" Approach 100
4.2.3.2.3 A "Policy/Mechanism Separation" Approach 101
4.2.3.3 Applying Separation of Policy and Mechanism to Interprocess Communication 102
4.2.3.3.1 Providing Flexible IPC Facilities 102
4.2.3.3.2 Support for Multiple Coexistent IPC Facilities 103
4.2.3.3.3 Providing Hardware Support for IPC 103
4.2.3.3.4 IPC for Decentralized Operating System Research 104
4.2.4 Related Work 104
4.2.4.1 Design and Implementation of Interprocess Communication Facilities 104
4.2.4.2 Policy/Mechanism Separation in Operating System Design and Implementation 105
4.2.5 Approach 106
4.2.5.1 Survey of the IPC Literature 106
4.2.5.2 Taxonomy of the IPC Design Space 106
4.2.5.3 Conceptual Framework for Representing IPC 107
4.2.5.4 Evaluation Criteria and Methodology for the IPC Primitives 107
4.2.5.5 Initial Collection of IPC Primitives 107
4.2.5.6 Trial Implementation of the Initial IPC Primitives 108
4.2.5.7 Evaluation and Iteration of the Initial IPC Primitives 108
4.2.5.8 Detailed Implementation and Evaluation of the IPC Primitives 108
4.2.5.9 Investigation of Hardware Support for the IPC Primitives 108
4.2.6 Contributions 109
4.2.6.1 Separation of Policy and Mechanism in Interprocess Communication 109
4.2.6.1.1 A Collection of IPC Primitives 109
4.2.6.1.2 Implementation and Evaluation of the Primitives and Policies 109
4.2.6.1.3 Estimation of the Suitability of a Policy/Mechanism Separation Approach 110
4.2.6.2 Applying Structure to the Interprocess Communication Design Space 110
4.2.6.2.1 A Taxonomy of Extant IPC Facilities 110
4.2.6.2.2 A System Model of IPC 110
4.2.6.3 Exploring the Use of Hardware Support for Interprocess Communication 111
4.3 References 111

5. DATE: A Decentralized Algorithm Testing Environment 115
5.1 Overview 115
5.2 Design and Implementation of DATE 115
5.2.1 Overview of DATE 115
5.2.2 Functional Specification 116
5.2.2.1 Overview of Facilities 116
5.2.2.2 A Scenario for Experimentation 116
5.2.2.3 Detailed Specification 117
5.2.3 A Sketch of the Implementation 120
5.2.3.1 The Structure of DATE 120
5.2.3.2 Implementation of Simulation 121
5.2.4 Present Status 123

6. Decentralized Computer Architecture 124
6.1 Overview 124
6.2 Separation of OS and Application Processing 124
6.2.1 Concurrency Techniques 124
6.2.1.1 Processes, Synchronization, and Communication 124
6.2.1.2 Virtual Memory and Protection 127
6.2.1.3 Device Interface 129
6.2.1.4 File System 130
6.2.1.5 User Interface 130
6.2.2 Generic Concurrency Techniques 131
6.3 Instruction Set Architecture Design 131
6.3.1 Introduction 132
6.3.2 Notions of Simplicity 132
6.3.2.1 Perceiving Distinct Qualities 133
6.3.2.2 The Utility of Complex Instructions 133
6.3.2.3 Designing Simple Machines 134
6.3.2.4 Complexity Migration 135
6.3.3 Importance of the Performance Aspects of Computer Design 135
6.3.3.1 What is meant by performance? 135
6.3.3.2 Performance vs. Other System Aspects 136
6.3.4 Ambiguous Performance Claims 136
6.3.4.1 All or Nothing 137
6.3.4.2 Fair Comparisons 137
6.3.4.3 Justification and Analysis 138
6.3.5 Architectures and Implementations 138
6.3.5.1 The Rules Have Changed 139
6.3.5.2 Departures From Purity 139
6.3.5.3 Moving to Higher Ground 140
6.3.6 Conclusion 140
6.4 Multiple Register Sets 140
6.4.1 Introduction 140
6.4.2 Evaluating Complex Instructions 142
6.4.3 Evaluating Multiple Register Sets 144
6.4.3.1 Performance Gains 145
6.4.3.2 Machine Support Requirements 147
6.4.3.3 Impact on Compiler Writing 148
6.4.3.4 Usefulness of Response 148
6.4.3.5 Language Effects 148
6.5 References 149

7. Interim Decentralized System Testbed 153
7.1 Overview 153
7.2 System Selection 153
7.3 Current Status and Future Plan 155
7.4 References 155

Appendix

A. Annotated Bibliographies A-1

A.1. Decentralized Operating Systems A-1
A.2. Interprocess Communication A-22
A.3. Hardware Support for Operating Systems Architecture A-66

B. Additional Work on Transactions B-1

B.1. Synchronizing Shared Abstract Types B-1
B.2. Transactions: A Construct for Reliable Distributed Computing B-35

List of Figures

Figure 2-1: ArchOS Development Steps 21

Figure 3-1: State transition diagram of the remote server and the user interface processes 77

Figure 6-1: An Architecture With Register Buffering 126
Figure 6-2: An Architecture for Concurrent Virtual Memory Management 128

1. Executive Summary

This document constitutes the Final Report (CDRL A004) for RADC Contract No. F30602-81-C-0297, Decentralized System Control. This report covers the period November 19, 1981 through March 31, 1984. It describes the results of our proposed work, which was performed by the Archons project during the contract period. It focuses on the studies of the fundamental issues of decentralized system control ranging from investigations of decentralized resource management principles to architectural support for decentralized operating systems. The report also includes a development plan of the decentralized Archons operating system (OS), called ArchOS. The description of a simulation environment of the decentralized algorithms, called DATE, and the current status of the Archons interim testbed are also described.

1.1 Archons and ArchOS Objectives

The Archons project is performing research on decentralized management of operating system level resources globally for an entire computer in which physical dispersal causes variable and unknown communication delays. We are interested in a very specific form of resource management decentralization: decisions are made by a team of equals who negotiate, compromise, and reach a consensus – the objectives are improved robustness and modularity compared with conventional unilateral resource management. Making decisons thusly, despite inaccurate and incomplete information about nonlocal state, involves nondeterministic computations. The scope of this management encompasses the operating system resources of all physical nodes in the computer, unlike a network which has communicating local operating systems. Failure atomicity requires a transaction facility in the OS kernel, but the usual serialization model of data consistency is insufficient for OS use – so, we have supplemented it with a relational one. The abstract types to be managed within a decentralized OS are different from the objects found in traditional databases, requiring innovative transaction techniques.

We are also interested in the architectural implications of our unique approach to operating systems: they arise in the interconnection structure and in the processor. Consequently, we believe that each node of the computer ought to consist of an application subsystem and an OS subsystem. The former may be arbitrary and heterogeneous but we are designing the latter ourselves. The OS machine (named Meta) at each node is an unusual functionally-oriented multiprocessor having an extremely maleable architecture to accommodate watever OS support mechanisms are desired. In addition, the hardware/software implementation tradeoffs are transparent to the OS programmer. There is a substantial experimental component in the Archons research, and the initial experiment is performed by the Archons interim testbed which consists of a set of Sun workstations interconnected by an Ethernet.

The objectives of the Archons project in general, and of its ArchOS operating system portion in particular, differ significantly in a number of ways from those of the other distributed system and distributed/network operating system efforts we are aware of.

The foremost of these dissimilarities has to do with our concentration on the special case of "distribution" which we term *decentralization* (explained further in Chapter 2):

- to explore the fundamental nature of making and carrying out decisions in a highly decentralized fashion
    - for resource management in general,
    - but for operating systems in particular;
- and thereby to facilitate the creation of
    - substantively improved computer systems in general (including uniprocessors and computer networks),
    - but especially a novel *decentralized computer* which can be physically dispersed yet which exhibits the optimality of executive level global resource management hitherto confined to physically concentrated (and highly centralized) uni- and multi- processor computers.

Our principles of decentralized decision making and resource management have wide applicability, from integrated man-machine systems, through application software, operating systems, and down to machine hardware. However, we are focusing on the OS levels (and below) for three important reasons:

- the OS is a constant beneath many changing applications -- this provides generality and lowers system costs by solving resource management problems once instead of leaving them to be solved repeatedly by the users;
- the degree and cost of successful decentralization above the OS depends on success at the OS levels and below;
- OS problems are almost always the most general, complex, and dynamic (elsewhere in a system the resources are usually more dedicated) -- consequently, solutions at the OS levels are more likely to be amenable for use at higher or lower levels, while the converse is much less likely.

Any system can be expected to have resources local to each node, but we are disregarding them in our research since managing them is so well understood by comparison with managing global resources.

1.2 Summary of This Period's Tasks

This section provides a very brief summary of each task described in this period's final report -- such information as:

- objective
- role in the overall Archons project plan
- research contribution
- current direction
- problems
- accomplishments
- future expectations.

1.2.1 Decentralized Resource Management Principles

Section 2.2 is devoted to the issue of seeking a new resource management paradigm which is *intrinsically decentralized* without the historical centralized biases and artifacts. This most fundamental issue underlies the entire Archons project. It appears to be philosophical and qualitative because the conceptual shift is dramatic and not yet amenible to analytical formulation. It also runs against the prevailing tide of intellectual inertia and career investment. Like many positions which espouse new philosophies, methodologies, etc., it can be best appreciated by those who have substantial experience with the older alternatives, the resulting problems, poor solutions, and attempted better solutions.

We are concerned with both *physical* and *logical* decentralization.

Physical decentralization does not mean simply spacial dispersal of nodes as in computer networks. Rather, it addresses the objective of knitting a collection of spacially dispersed nodes into a single computer. This requires a completely new kind of operating system: one which has physical distance *inside* it. This results in variable and unknown communication delays which are significant with respect to the rate of system state change. New concepts and techniques for resource management are called for -- these include in particular:

- replacement of the "garbage-in/garbage-out" point of view with a "do the best with what you can get" one (i.e., accommodating and even taking advantage of indeterminism);
- replacement of the "processing-oriented" point of view with a "data-oriented" one, in which the principle goals are maintaining the *consistency* of data objects, and the *correctness* of the actions carried out on those objects.

Logical decentralization in our case is interpreted as being in a region diagonally opposed to the origin (representing maximal centralization) of a 7-dimensional space of resource management. The initial version of this model was developed on an earlier contract. The degree of logical decentralization we aspire to involves each decision being made by negotiation, consensus, and compromize among team members, each of whom is equal in authority and responsibility but has a different perspective due to being resident on a different node of the computer.

This task has reached a plateau; the first order technology requirements are clear, and are being pursued in other tasks. Foremost among those are an information-theoretic technique based on team decision theory, and a complementary heuristic effort; both need a resource management context, and we have selected assigning processes to processors (a context which is valuable from a system point of view as well). Feedback from those are causing the philosophical framework to be updated and further developed.

1.2.2 ArchOS

ArchOS is the name of the Archons project's initial decentralized global operating system. In the preceeding years of our research, we have focused on philosophy and technology, while keeping an informal vision of the Archons system and ArchOS operating system in mind. Now that philosophy and technology are mature enough for us to begin solidifying that vision. This will be one of our major activities during the follow-on contract periods, beginning with a document defining our goals and objectives, together with the requirements, for ArchOS. That will be followed by a functional specification, design, and implementation (on an interim testbed of Sun's). Section 2.2 summarizes the major objectives of ArchOS, and Section 2.3 discusses our development plan for ArchOS.

1.2.3 Predominant OS Functions

This effort is intended to identify the OS functions most in need of architectural support: those which consume many processor cycles as a result of their complexity, frequency of execution, or fast response time. Ideally, this task would be focused on decentralized operating systems such as ArchOS. Obviously this cannot be done at present since ArchOS is not far enough along, so a first approximation is being based on centralized operating systems; this will at least help formulate performance benchmarks for the architecture studies discussed in Chapter 6. We have recently de-emphasized this effort due to the greater need for ArchOS design manpower, and to the difficulty of acquiring the intended information about other operating systems.

1.2.4 Transactions

We have two different tasks under way on atomic transactions.

- One, described in Section 3.1, is oriented toward decentralization and programming modularity. To the best of our knowledge it is totally unique research in that it utilizes only *transaction syntax* as a basis for consistency of nonserializable transactions. The concepts in this phase of the effort have largely stabilized, and most of the work has been formal -- since the notions of data consistency and transaction correctness underlie our entire system approach, it is incumbent upon us to provide a theoretical foundation to insure their validity. We were first able to analytically prove that no alternative approach using *semantic* information could provide better concurrency of actions; then we proved that none could even do as well. The next stage of this task is to deal with fault recovery.

- Of course theoretical research necessarily makes simplifying assumptions; in actual systems, complex and informal tradeoffs are required. For this reason, we have a second transaction task taking a different tack, as shown in Section 3.2. It is less decentralized in that it uses *global semantic* information, aligning it with the few other research efforts we know of in the area of nonserializable transactions. It concentrates more on complex abstract data types, and has already made fault recovery a major theme. Progress here is slow but steady.

1.2.5 Interprocess Communication

IPC is an essential element in a distributed system, and even more so in our style of decentralization. We expect to experiment with a variety of new IPC concepts and facilities for ArchOS in order to obtain the benefits we seek. This necessitates a design methodology which permits rapid, easy redesign and reimplementation. Policy/mechanism separation has for some time been considered in this regard, and has been attempted in limited OS contexts such as process scheduling. However, we believe that it has not been approached properly, and consequently it has been far less than successful. Policy/mechanism seperation has never been even attempted in IPC which is far more complex than, for example, processor scheduling or memory management. We expect to make important contributions to both the fields of programming abstractions and interprocess communication. Some of the biggest intellectual hurdles have been overcome, and constant progress is assured through the duration of the next contract period.

1.2.6 Decentralized Algorithm Testing Environment

Chapter 5 outlines DATE, a discrete event simulator for performing experiments with decentralized algorithms on VAX/UNIX. Even with our Sun-based Interim Testbed operational, a proper simulation facility offers advantages which would be more expensive to achieve on the testbed: e.g., stimulus and instrumentation mechanisms, reconfigurable topologies, alterable communication subnet characteristics. DATE is now completed and has been installed as well at NOSC in San Diego.

1.2.7 Decentralized Computer Architecture

Decentralized resource management should not be considered merely an OS matter, but rather a system matter. Our operating system is sufficiently unusual that it suggests reconsidering the architecture of the processors and their interconnection[1].

- We have been looking at two aspects of the processor architecture topic: seperating each node of the computer into an OS part and an application part; and the design of the OS part. This allows existing machines and application software to be retained, while at the same time the OS processor can be designed expressly to facilitate decentralized resource management. Concurrency of OS and application execution also improves system performance.

  - The first aspect began with an extensive evaluation of the literature on architectural support for operating systems; substantive subsequent progress remains to be made this academic year.

  - The second aspect began designing the OS machine architecture, but then turned to developing an improved methodology for doing so. So little basic scientific and engineering perspective is normally used to design and evaluate machines that we felt compelled to intervene in that respect. Our desire was only to do a good job on our own machine, but by coincidence the general topic became one of the hottest in computer architecture (i.e., the "RISC/CISC" controversy). Our methodology contribution had unexpected impact in this controversy, with side effects on the community's attitudes and perceptions.

    This aspect has two major thrusts going forward: separation of the effects of register structure from those of instruction set complexity; and developing a systematic approach to functional migration (e.g., from software to microcode or hardware). While these subtasks will be important as ends in the field of computer architecture, to us they are primarily means to the end of designing our own OS machine named Meta. Both subtasks will be completed in this academic year, allowing progress on Meta to resume.

1.2.8 Interim Testbed

The Archons project has two testbed facilities in its plans: an interim one based on commercially available hardware and software; and a later one of our own hardware and software design. The former will support experimental research with both decentralized algorithms (e.g., process/processor binding, IPC, transactions) and decentralized operating system structures. This experience will eventually lead to the development of a testbed which will allow experimental research on not just design but also implementation of both software and hardware.

Chapter 7 provides an overview of the current Interim Testbed system. The main software requirements of the testbed were having both the Unix operating system and a lower level executive (in our case, BBN's CMOS). In addition, the hardware was required to employ the Multibus backplane and a 68010 processor.

---

[1]This work is sponsored by the U.S. Army Center for Tactical Computer Systems.

Thus, we selected the Sun workstation for both technical and administrative reasons. At this point, we have reached a state where the testbed has a stable minimal hardware and operating system configuration and completed the installation of CMOS on our Sun workstations. The explanation of the system selection and the current status of the Archons testbed system are included. Our plan for the further development of the testbed is also described.

2. ArchOS: A Decentralized Operating System

2.1 Overview

This chapter outlines how the decentralized resource management concepts we have been, and are, creating can be utilized to construct an operating system which is radically different from current practice in both principle and behavior. This initial operating system is named ArchOS, and will be an experimental existence proof. Not only its objectives, but also a preliminary development plan, are described.

2.2 ArchOS Objectives

2.2.1 Background

Several of the principals in the Archons project have been designing and implementing a variety of distributed systems, primarily in military/industrial R&D contexts, for as long as 14 years (and continue to do so as consultants and corporate employees). These systems have been relatively innovative in many respects, yet as conservative as need be for transfer of the technology to product environments without excessive risk. Experience with these research prototypes and their progeny products exposed us to many of the critical problems of physically and logically distributed systems, and to the frustrating limitations of trying to adequately solve these problems with approaches from conventional "unicentric" computers and "polycentric" computer networks. As is all too often the case, many of the specific details of these systems and our experiences remain under corporate proprietary and military classification shrouds. However, they clearly manifest themselves in the the objectives and directions of the Archons project and its ArchOS operating system effort.

A university normally imposes few (but not necessarily no) "product" pressure constraints -- the amount depends on: the extent to which a project is generating a stable facility for users, versus research results per se; the project sponsor(s), and contractual relationships; etc. Being in an academic environment which is suitably oriented and equiped for large scale experimental hardware and software research (such as CMU's Computer Science Department), and having appropriate DoD and industrial sponsorship, we could have chosen to immediately apply the lessons we have learned from the work of ourselves and others to the design and implementation of an adventurous distributed system and operating system (OS); we believe that it would have made *significant contributions to the field* (and that we would have greatly enjoyed ourselves).

Instead, we chose to embark on a much more ambitious, longer term, and potentially higher payoff research effort:

- first seeking visionary new resource management paradigms which are as *intrinsically "decentralized"* as we could conceive of;

- then employing these as the foundation of an experimental decentralized OS;

- and finally utilizing the resulting OS concepts and techniques to perform hardware/firmware/software implementation tradeoffs which lead to a second generation OS, and the hardware design of our own optimal architecture for it --

we view decentralized resource managment as a *system*, not just a software, effort [Jensen 81a]. (This raises the issue of whether such a machine is, or should be, a complex, as opposed to a reduced, instruction set computer; our choice is neither, which we have commented on in [Colwell 83].)

Evolution is generally appropriate as the primary mode of computer (and other) system development, but it should be performed with much careful thought. Almost all work on "distributed" systems in general, and "distributed"/network operating systems in particular, has been evolutionary to an extreme -- most of the resource management concepts have been simple adaptations of *centralized* ones, burdened by inappropriate and even counterproductive artifacts. The ineffectiveness of constructing airplanes which fly by flapping their wings was recognized early; but corresponding realizations about distributed systems have largely not yet taken place, as we have argued for several years (e.g., [Jensen 76]) and briefly review in Subsection 2.2.1.

Exploring new frontiers, especially on a large *systems* scale, can be not only exciting and stimulating, but also frustrating, tiring, and even dangerous. A paramount source of the unpleasantness is the extensive duration such an effort can entail:

- a system design is broad -- the number and intricacy of its interacting problems rises exponentially with size;

- a particularly formidable and key problem may consume a great deal of time;

- an additional interval may elapse before it is prudent to disclose one's solution to a particular problem.

One reason for the disclosure delay is that an unconventional and perhaps controversial result might be viewed skeptically as more of a conjecture if it is not substantiated by evidence, which can often require concommitant results or extensive experimentation (neither of which may be complete at that time). In addition, government funding for building large scale systems is rare and often competitive, most of the aspirants commonly being corporations; the satisfaction of seeing your ideas appreciated through their appearance in someone else's proposal may be too high a price to pay for losing a unique opportunity to implement and experiment with those ideas yourself. (In some of our own cases, a certain degree of reticence has doubtless been instilled as well by extensive careers in military/industrial research where the incentive to publish ranges from marginally positive to strongly negative.)

Partially as a consequence of these delay effects, social pressures may come to bear -- one's peers (or management) may form misconceptions about the nature, feasibility, quality, or quantity of the research. Employment security of the researchers may not be well established, adding an element of personal risk to the venture. Student contributors are pursuing degrees and expect to graduate in a bounded period of time.

Students also have academic (and social) obligations which detract from the amount of effort they are willing and able to commit to research, which makes fulltime professional (e.g., post-doctoral) personnel almost essential on large scale experimental projects. But supporting professional researchers may be viewed by some as contrary to the pedagogical imperative of academia.

A lengthy research project must also be able to adapt to relevant (supportive or not) results from other efforts; assimilation and agility help prevent obsolescence.

And ultimately, there is always the nonzero possibility that maps which claim "Here be dragons" or people who assert that "You will fall off the edge of the world" may be (at least partly) right -- in the quest for new paradigms, negative results are valuable, frequently more so than positive ones.

We felt that the Archons principals had the requisite experience, insight, self-assurance, physical and emotional endurance, security, intellectual environment, and physical facilities to accept and conquer the "insurmountable opportunities" of this challenging research project. (Our feelings in these respects do continually vary through the course of our research.)

2.2.2 Decentralized Resource Management

There currently seems to be little common understanding about what "distributed" decision making or resource management means; this is one of the reasons that the "distributed" and network operating system in the literature and laboratories are so very conceptually and functionally disparate. We have chosen to use the term "decentralization," and to attempt to rather carefully (albeit not formally) define what we mean by it.

"Centralized" and "decentralized" are not usefully viewed as a dichotomy, but rather as the endpoints of a continuum -- indeed, as diagonally opposed vertices of a multidimensional space. We are not under a misconception that extreme decentralization is necessarily advantageous in all ways and under all circumstances. However, our experience (both prior to and subsequent to the initiation of the Archons and ArchOS research) provides cogent arguments and concrete evidence that movement away from the heavily populated highly centralized subspaces can be invaluable. At least in some applications we are familiar with, such as supervisory real-time control (e.g., combat platform management and factory automation), more decentralization of resource management offers improvements in certain system attributes like robustness and modularity. In

order that we, or any designers of a particular system, be able to *scientifically* position ourselves well in this space from maximally centralized to maximally decentralized, far too little knowledge exists today about its decentralized boundary conditions.

It became apparant to us that these issues ought to be dealt with explicitly and systematically from the ground up, not in the prevalent ad hoc adaptive (albeit safer and faster) way, if the many attractive promises of a physically dispersed computer were to be realized. Thus, we launched an extensive search for the limits of resource management decentralization, divided into two areas: logical and physical. (Computer scientists sometimes imagine incorrectly that logical things are innately more conceptually interesting than are physical things; the opposite seems true to us in this case.)

2.2.2.1 Logically Decentralized Resource Management

One of our first steps was to create a conceptual model of the space of *logical* decentralization of decision making; the most detailed of its incarnations can be found in [Jensen 81b]. It can be applied at different levels of abstraction, from one instance of one decision about one resource, through all instances of all decisions about all resources. Our model is germane to the management of local or global resources. In it, decentralization is founded on *multilateral* management; not, for instance, on the more common theme of resource or functional partitioning (which leads to autonomy as maximally decentralized, which we reject). Our model expresses the degree of decentralization as being determined by several factors, crudely summarized as follows:

- the percentage of resources involved;
- the percentage of decision makers which participate (depending on the level of abstraction under consideration, functional partitioning and successive techniques such as round robin may be placed at the centralized end of this axis);
- the extent to which all decision makers must become involved before *a* decision has been completed (note that resource partitioning and functional specialization are highly centralized by this metric);
- the degree of equality of decision maker authority and responsibility (this axis places a premium on peer relationships, in contrast with the ubiquitous hierarchical ones).

To this version we subsequently added a negotiation axis, which we summarize in print here for the first time.

At the minimum, more centralized, end of the negotiation axis, each decision is made by a collection of entities which work as a "team" to move the overall system toward its goals. Any team member is allowed to make certain (not necessarily fixed) decisions without necessarily gaining the concurrence of the other team

members: these decisions may be constituent subdecisions or different instantiations of the same decision (this difference is represented on other axes of the model). Any team member may seek information which will improve the quality of its decisions [Marschak 72]. A well-known example of team decision making is routing in the ARPANET communication subnet [Ahuja 82].

At the opposite, more decentralized end point, each member in a collection of decision makers develops hypotheses (deductions and assumptions) with associated probabilities -- these may be based on some form of partitioned competence (designated elsewhere in the model) or disparity of information (which may again be a logical factor, or a physical one as discussed in the following subsection). To make a decision, members exchange these, reason about and modify them, making compromises as necessary, and in this way enhance the marginal viewpoints to a more global view. This activity must somehow converge to a single consensus decision, perhaps by a formal method such as that of DeGroot [DeGroot 74], or by heuristics such as inference rules and algorithms -- the latter are employed by some areas of artificial intelligence such as problem solving and expert systems (e.g., the HearSay system [Erman 73], [Erman 79]). (Note, however, that HearSay was quite centralized by our standards: logically, because the knowledge sources were functionally specialized; and physically, due to the shared global state "blackboard".) This technique is somewhat similar in spirit to the "divide and conquer" approach of algorithm design but lacks the optimality of the full Bayesian method, because the joint information has been sacrificed. That is, the various inter-dependences are only approximated by the indirect approach of reaching a consensus.

We are interested in the conditions under which different degrees of logical decentralization according to our model offer how much of which attributes, and the tradeoffs involved, in managing global resources. But the region of primary interest to us in this multidimensional space of logical decentralization is where each global decision is made multilaterally by a group of peers through negotiation, compromise, and consensus.

According to our view, most resource management is highly logically centralized, even in the myriad network and distributed operating systems we are aware of.

2.2.2.2 Physically Decentralized Resource Management

We have long argued that the important benefits of having *system-wide* resource management at the *operating system* level, routinely provided by a computer, are not available to many systems -- the reason is that those systems consist of multiple nodes which must be physically dispersed (for functionality, reliability, and logistical reasons).

Unfortunately, operating systems as presently conceived are highly and inherently centralized in several critical respects. Perhaps most importantly, they are based on some very strong premises about time -- e.g.,

that communication delays due to physical dispersal within the operating system are practically negligible with respect to the rate at which the system state changes (note that the same effect can occur on VHSIC/VLSIC chips). This leads to the presumption that it is possible (and even cost-effective) for all processes to share as complete and coherent a view of the entire system state as may be desired (e.g., that a single global ordering of events can be established). Another class of centralized operating system premises has to do with the types, frequencies, and effects of faults, errors, and failures. Both the time and fault premises are rational given the historical evolution of operating systems in the context of shared primary memory (i.e., uniprocessors and multiprocessors). Unfortunately, many of these premises go unstated (e.g., in operating system texts and papers), and are either forgotten or assumed to unquestionably always hold.

Our focus is on achieving the global executive level resource management for a physically dispersed system to be a *computer* in the same sense that a uniprocessor or multiprocessor is. However, we are not restricting ourselves to virtual uniprocessors -- is it frequently beneficial (e.g., improved fault recovery and performance) for some image of the composite and decentralized structure of the software or hardware to (occasionally or optionally) be made or left visible to the user.

Presently, Archons appears to be essentially alone in stressing unification at the operating system levels; the dominant theme in distributed system projects today is "autonomy." The only popular alternative to conventional centralized computers is *computer networks*. A conventional generic computer network can be characterized as follows:

- each computer is (functionally and often administratively) autonomous with its own local, centralized operating system;
- all the computers are connected to a communications subnetwork;
- each computer has network server utility software (for transport protocols, naming conventions, and the like) sufficient for them to do resource sharing (e.g., file transfers, mail, virtual terminals);
- there may be higher layers of software for specific applications (e.g., banking, military C^3), perhaps giving the users some unified perception of the system.

A network normally is supplied with a so-called "network operating system", which tends to simply be the collection of network server utilities. Historically it has been constrainted to being a guest of the local operating systems; recently, more indigenous (and thus more effective) network operating systems are developing (e.g., [Rashid 81]). A few recent networks and their network operating systems aspire to eventually make gradual movement in the direction of greater operating system coordination (e.g., [Spice 79]).

Many applications need nothing more than long-haul resource-sharing or value-added networks, or local

area networks of personal workstations. But not in the cases of concern to us: the very complex problems of achieving system-wide resource management are forced up to the user level where they are more difficult (having less access to lower level resources and receiving little assistance and perhaps even resistance from the local operating systems); and where they must be solved repeatedly if there are multiple users, instead of once by the system designers. The unsurprising consequence is that these applications with substantive state change/visibility ratios must suffer: because of the solopsistic local operating systems, *system* robustness is poor, modularity is compromised, performance (e.g., concurrency) is reduced, and total system cost is increased.

We fought with the dilemmas of this dichotomy between computers and computer networks during the past decade of our experience designing distributed systems and realized that having a physically dispersed computer requires a functionally singular operating system (as opposed to a network of independent private operating systems). The primary obstacles to be overcome are that:

- communication *within the operating system* is inaccurate and incomplete with respect to system state changes;
- and the types and effects of faults, errors, and failures encountered in multinode physically dispersed systems differ significantly *in both degree and kind* from those in single node systems -- this is even more pronounced in a decentralized computer than in a computer network.

3.1.2.1 Accomodating Imperfect Information

High degrees of physical decentralization imply that resource management decisions routinely must be "best effort", based on imperfect quantity and quality of information -- the virtually ubiquitous "garbage in, garbage out" characterization of computers is unrealistic and cannot be tolerated in a physically dispersed multinode computer. This perspective is somewhat familiar *above* the OS levels (e.g., in certain artificial intelligence work), and *below* them (e.g., in dynamic communication packet routing). But *at* the OS levels (as in most software), it is a foreign outlook which is incompatible with the current state of the art. Consequently, new problems have to be solved in the design of the decision algorithms, such as: picking thresholds of result acceptability, and specifying them to the decision makers; determining what "value" the completeness and accuracy of information utilized contributes to the "quality" of a decision result.

In thinking about these issues, it becomes clear that while the logical and physical aspects of decentralized decision making are conceptually distinct, they strongly interact. For example, the decision convergence time may include acquiring suitably valuable quantity and quality of information, as well as negotiating.

An unavoidable characteristic of a physically dispersed machine is a significant increase in the indeter-

minism of its behavior. The centralized mind set is that not only ends but also means at all levels ought to be entirely deterministic; it is normally affordable to closely approximate this in a centralized machine, and instances to the contrary are dealt with in various ad hoc fashions. Our position is that considerable indeterminism is the *normal* case in decentralized resource management, and can be exploited to advantage (e.g., improved robustness and performance) rather than merely tolerated. Dynamic packet routing demonstrates our position, and we have done so (transparantly to the users) in a network operating system ([Sha 83]).

3.1.2.2 Faults, Errors, and Failures

In a physically dispersed multinode system (whether network or computer), reliability problems are worse than in nondispersed and uninode systems, particularly when considered in light of the imperfect information issues discussed above. For example, concurrency control and failure atomicity become vastly more complicated, far beyond the realm of current centralized operating system conceptions. Computer networks have more relevant technology in this respect, but most of it is actually inspirational rather than directly transferable to a decentralized OS and computer.

We address failure management and recovery within an operating system by thinking of the OS state as a special kind of distributed database which is approximately replicated at each node. This suggests that an atomic transaction facility for use by the OS (and perhaps by higher, e.g., application, levels) be incorporated in each instance of its kernel ([Jensen 81c], although we made this pivotal design decision in 1978 as a result of several enlightening discussions with Gerard Le Lann about his distributed database concurrency control research). As a consequence, three classes of significant research issues arise.

One is that both the services and structure of the OS ought to be substantively affected by the availability of atomic transactions as kernel primitives. This is essentially virgin territory: the few approximations to atomic transactions at the OS level have been ad hoc; in fact, they have not been explicitly viewed, designed, and exploited as transactions.

Secondly, the overhead (especially communication) of atomic transactions, which is always a concern, becomes of paramount importance at the OS kernel level. We have determined that one can achieve great acceleration without degrading flexibility with thoughtfully crafted hardware *mechanisms* at the disposal of software modules which establish the desired policies.

The third class of research issues has to do with the need for insightful reconsideration of atomicity itself. While the inclusion of atomic transactions in our OS was inspired by their contributions to conventional database systems, our transaction facility differs radically in several respects from those used in that context. Examples include the following.

The conventional serializability theory of concurrency control includes the assumption that consistency and correctness result from a single transaction executing alone; this leads directly to the same result for all serializable schedules. An advantage of serializability is that it is completely general and works without requiring knowledge about either the database or the transactions; but a disadvantage is that it cannot exploit such knowledge which may be available in any specific case (such as in an OS). The cost of this generality is that serializability can exclude consistent and correct schedules which provide higher concurrency than those it permits. Database researchers have begun to study *nonserializable* consistency control methods in search of greater concurrency, but a major difference is that a transaction can no longer be regarded as if it were executing alone. Consequently, the consistency and correctness properties of nonserializable scheduling rules do not follow automatically, they must be proven. So that every attempt to utilize nonserializability is not burdened with inventing its own rules and proving their properties, a formal theory of nonserialzable concurrency control is needed. Because it is already known that serializability theory provides the highest degree of concurrency possible when using only the classically defined transaction syntax, some additional kind of information is required if nonserializabilty is to perform better. Researchers other than ourselves seem to be focused exclusively on exploitation of transaction *semantics*, developing syntactic structures to support programmers' specification of their own application-dependent scheduling rules. All programmers involved with any given database must understand the details of each other's transactions, and every programmer is responsible for the consistency and correctness properties of his own rules. Such extensive use of global transaction semantics (e.g., the "break point specifications" in [Lynch 83] and "lock compatibility tables" in [Schwarz 82]) allows very high degrees of concurrency, but appears to limit this approach to rather static and specific situations. *Modularity* is recognized to be extremely valuable in software engineering generally; we consider it a critical attribute in transaction-based distributed computations, particularly decentralized operating systems. Therefore, we have created a different and more decentralized theory of nonserializable concurrency control which seems improved with respect to modularity; when a programmer schedules one of his transactions, he need know only the agreed upon transaction syntax, the details of his own transaction, and the consistency constraints of the database subset affected by this transaction. We define a new transaction syntax, called *compound transactions* (of which nested transactions are a special case), and its associated *generalized setwise serializable* scheduling rules (of which serializability is a special case). Our schedules are *complete* in the sense that for any consistency and correctness preserving schedule, there exists an equally consistent and correct setwise serializable schedule which provides at least as much concurrency.

An important implication of our different approach to transactions is that failure management and recovery must be re-evaluated. The usual notion of failure atomicity is drawn from serializability theory, where a transaction cannot be commited until all its actions are successful and in stable storage. Higher concurrency can be achieved by determining conditions which permit a transaction to commit completed steps before the

end of the transaction. Our concept of *failure safety* is based on such conditions. In [Sha 84], we formalize our theory, and its properties of consistency, correctness, modularity, optimality, completeness, and failure safety.

Other major departures we must make from normal atomic transaction facilities include:

- Instead of being above an OS with the corresponding functionality to draw on, our transaction facility is *beneath*, inside the OS kernel. This affects the facility design substantially.
- Rather than handling simple database objects such as records and files, it must accomodate the far more complex, abstract, and dynamic data types found in an OS.
- An object is not necessarily located at a single node -- a single instance of it may be physically dispersed across multiple nodes.

Note that the work outlined above has applicability beyond our motivation to enable the creation of a logically and physically decentralized operating system (and computer) which is extremely reliable and modular.

2.2.3 Other Objectives

In this subsection, we cover some salient objectives of distributed system/OS projects which *are*, and *are not* factors in the Archons and ArchOS effort -- this will help familiarize the reader with our research and distinguish it from the other work in this general field.

2.2.3.1 Research Per Se Versus Facility Development

The Archons system and its ArchOS operating system are vehicles for our *own research* in decentralized resource management -- this has three major ramifications:

- First, and most important, is that projects (e.g., Spice here in the Computer Science Department, and many others in progress elsewhere) which are intended to result in a general computational facility necessarily have shorter term schedules, and thus scope and risk constraints far more conservative than ours.
- Second, we have no desire to be compatible with anything; in particular, ArchOS is not compelled to present its users with a UNIX interface.
- Finally, the Archons and ArchOS hardware and software are privately owned and operated by the Archons project rather than by the Computer Science Department's research facilities group (although we are very grateful for their kind cooperation and assistance). While we consequently lose the valuable committed support of that group, we are also are not subject to Department logistical policies regarding permissible hardware and software.

ArchOS is an experimental prototype which will serve, among other things, as an *existance proof* that our new resource management paradigms are valid and that is possible to base an OS on them -- if obliged to, we will treat cost-effectiveness and even feasibility as almost second-order effects.

We expect to include only a subset of the services usually associated with an OS: those for which we have designs or implementations that meet our research objectives (e.g., process to node binding); or those which we need ourselves (regardless of how centralized or decentralized we do them). Everything else will be considered dispensible.

The design and implementation of ArchOS will be in a constant state of flux, and will not present stable facilities to its users. Some of our research sponsors will possess copies of ArchOS, but we are obviously unwilling and unable to support them in the field; at least two of these sponsors expect to maintain some version of ArchOS themselves, and IBM has expressed a willingness to consider providing ArchOS support for the others.

2.2.3.2 Large Scale Experimental Computer System Research

An unfortunate limitation of most U.S. university computer science (CS) departments is their inability to conduct large scale experimental research. NSF and other national (and even a few state) government agencies, together with some industrial corporations, are trying to help remedy this, but still very few CS departments have the requisite facilities and conducive environment. This is particularly true for *computer*, as contrasted with software, systems; even most EE departments suffer in this respect. Given the facilities and environment, there is still the choice of research style to be made (see Section 2.2.1). For want of either opportunity or desire, most computer scientists in the computer systems area do not design, implement, and experiment with, large scale systems. Numerous distributed systems of various types are being constructed, but many are small in one or more respects, and virtually all are entirely software efforts utilizing existing commercial computers (typically ranging from LSI-11's to VAX's) and interconnection hardware (usually an Ethernet). It is interesting that except for certain military systems, this characterization holds for industrial as well as academic distributed systems.

One of the objectives of Archons which most differentiates it from other distributed system research is our willingness, desire, and indeed determination, to reflect our unconventional OS in the architecture of the hardware (both processors and interconnection) it runs on.

Initially, we are employing the CS Department VAX's plus our own interim testbed for algorithm experiments, simulations, and software development. A project facility was required in addition to the department one because:

- some of our concept and algorithm experiments would be distorted by system sharing (e.g., Ethernet traffic);
- other experiments would be interfered with by the VAXs' UNIX operating systems -- we need the freedom to substitute a simple executive;

- ArchOS is native (i.e., executes on the bare hardware).

There are two reasons for beginning with interim hardware: the requirements for the Archons decentralized computer hardware are primarily generated by ArchOS, which has not yet progressed far enough for its underlying needs to be clear; and even if we knew now exactly what those needs were, the hardware effort is itself very large scale and will require considerable time to complete.

Our interim testbed is deliberately conventional network technology. The selection requirements for it were: off the shelf computing and connection hardware to ensure immediate availability; the use of a Multibus backplane, so that we could do our own system integration and modifications as desired (e.g., multiprocessor nodes, special support hardware boards); Berkeley UNIX, for compatibility with the CS Department VAX's, and for its networking and other enhancements; the processor being a 68010, for software development tools and other software availability. These led uniquely to the Sun Microsystems, Inc. products.

Our eventual Archons decentralized computer will be highly unconventional in essentially every respect. For example, each node consists of an application subsystem and a resource management subsystem -- the user programs execute in the application subsystems (which may be heterogeneous and whatever the application calls for); ArchOS executes in the resource management subsystem which is based on a very unusual machine of our own design (named Meta), optimized for decentralized resource management and the corresponding attributes we seek.

2.2.3.3 Application and Attributes

The application environment of principle interest to Archons and ArchOS, at least initially, is "upscale" supervisory real-time control -- e.g., military combat platform management, large scale industrial factory automation. It is rare for an academic project to focus on real-time applications: faculty and students have little exposure to, and thus understanding of, this class of problems; and there is sometimes a sociological tendency to avoid working directly on projects of military significance (even though nearly 100% of academic computer science and engineering research is funded by the DoD, and is useable by the DoD regardless of who funded it). The real-time control environment offers both simplifications and complications. The former is that such systems are typically dedicated function, implying that at least some of the resources can be managed in a more static style than possible in general purpose systems; at any particular state of the OS art, this may make the difference between being able to perform a function in a highly decentralized manner and not (explaining why so many of the most interesting distributed systems have been, and continue to be, found in the military real-time control field, albeit hidden from public sight). The latter arises from the essence of real-time control, that resource management must be time driven -- so mere existance of the functionality doesn't suffice, it must meet deadlines as well. We find this combination of opportunity and challenge ideal,

and in fact irresistible when combined with the eagerness of military customers to not just fund but experimentally *apply* innovative systems such as we want to create.

While it may seem contradictory to our selection of real-time control as a application environment, performance (especially in the throughput sense) is not one of the more important properties we seek to attain with highly decentralized operating systems and computers. The contradiction is an illusion, because we are designing the *response time driven precept* as a fundamental characteristic into our resource management principles, algorithms, OS, and system. The actual *magnitudes* of the deadlines any particular design or implementation can handle is of less significance to us, and will be the subject of subsequent performance optimization work.

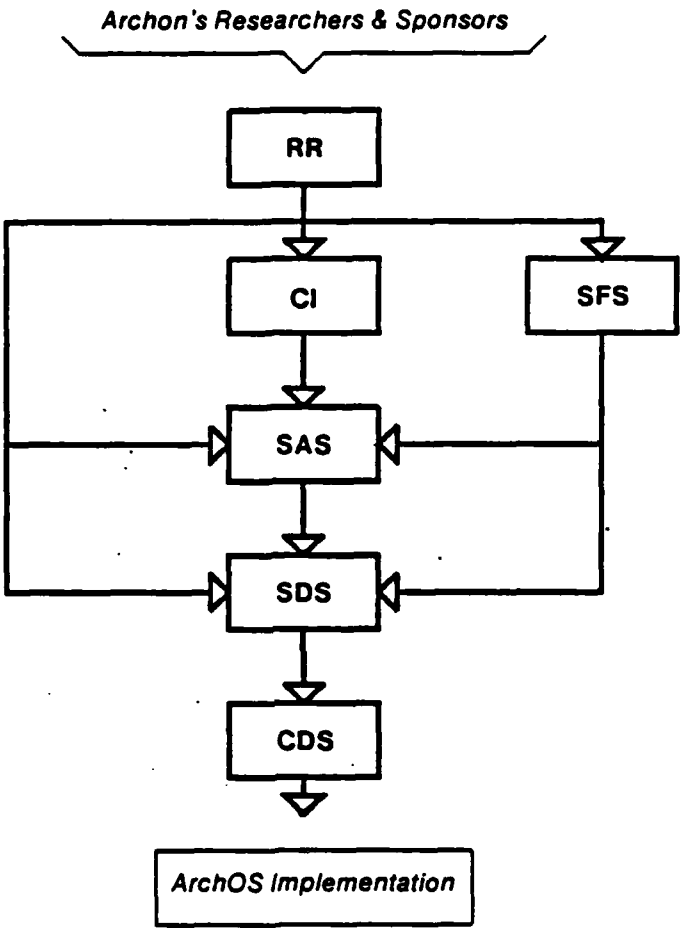
Moreover, we perceive that performance of most systems will improve automatically (and rapidly) with advances in semiconductor technology. But we can expect little if any assistance from semiconductor technology in areas of equal or greater importance, such as fault tolerance and modularity -- *these* are what computer systems research ought to attempt to improve. The common bias toward performance without acknowledging what is being traded for it (e.g., the reduced instruction set computer controversy) is not because performance is so much more important than other system attributes, but in our opinion because it is so much easier to attain and measure.

2.3 ArchOS Development Plan

This subsection discusses some of the methods we are employing and documents we are producing in the course of developing the initial version of ArchOS. The overall progression is illustrated in Figure 2-1. While this methodology isn't as elaborate as good industry practice, it appears to be far more extensive than that conducted in other academic distributed system projects. This reflects not only the industrial background of several of the Archons principals, but also the fact that the scope and complexity of the ArchOS research demands careful program management. We believe that our software specification techniques make some novel technological contributions of their own.

2.3.1 Research Requirements

An OS development effort is usually launched with the presentation of a Requirements Specification document, that emphasizes the performance and resource utilization aspects of the OS (e.g., response times and storage restrictions), some of the key internal structural requirements (e.g., a tree structured file directory system), and certain characteristics of the users' interface. The development then proceeds to optimally meet these requirements by any means subject to various program management constraints.



RR:	Research Requirements
CI:	Client's Interface Specification
SFS:	System Functionality Specification
SAS:	System Architectural Specification
SDS:	System Design Specification
CDS:	Component Design Specification

Figure 2-1: ArchOS Development Steps

We felt that ArchOS needed such a directing document, but one with a completely different emphasis. It must confine the development of ArchOS so as to assure that:

- ArchOS will satisfy the *research* objectives of the Archons researchers and sponsors. These involve understanding the characteristics and costs of decentralized (as we have defined it) operating system resource management. They do not involve client convenience or performance optimization.

- ArchOS will have no centralized implementations of the OS functions (at any level)

    - because they are familiar, or

    - because they are "obviously the best" (usually in a performance sense), or

    - because they are allowed to creep in unintentionally.

- ArchOS will neither build upon nor offer any mechanisms that are based on unfounded assumptions carried over from our experiences with centralized systems.

- ArchOS will provide complete internal observability to the experimenters (but not to the application level clients). As a vehicle for experimental research, ArchOS must readily reveal the kinds of data that make experiments meaningful.

- ArchOS will support change of both facilities and implementations. So little is known about the nature of decentralized resource management that we must anticipate the need to implement and evaluate alternative approaches.

Our Research Requirements document captures these notions, enumerates specific sponsor requirements, and supplies check lists to be applied during the review of each subsequent work product. We are not calling this document a specification because we feel that it will probably be impossible to measure the degree of compliance to many of the items that it contains.

2.3.2 Clients Interface Specification

We are using the Research Requirements to establish the clients' view of ArchOS, which is being recorded in the Clients Interface Specification document. This document defines the entire external interface available to a client process, and specifies ArchOS's behavior as observed at that interface. Because ArchOS is expected to manage all the system (global) resources, this is also the clients' view of the decentralized computer system.

Having the clients' interface possess features that would make it convenient to be used interactively, by a person, is a low priority concern to us at this time. Our driving concern is for the interface to be rich enough to allow the needs of the clients to be completely conveyed to ArchOS. It is meaningless to use a phrase like "best-effort", if it is based on pre-determined notions of the ArchOS designers instead of on information that can only come from the application using the system.

We believe that an OS that forces the client processes of a system to know the details of the system's resources places all responsibility for reliability and availability on the application. That is, if the clients have to base their OS requests on the initial configuration of hardware, then they similarly must base these requests on the current state of the system when it is running in a degraded mode. Even worse, it would force them to account for the fact that the "current state of the system" could look different to each user process, because of unknown and variable communication delays. This illustrates why we must treat phrases like "current state of the system" as being (at best) probabilistically defined.

The clients' view must not depend upon the structure of ArchOS, the mechanisms that implement ArchOS, or the internal strategies used by ArchOS. Allowing such dependencies would necessarily corrupt the validity of data collected in comparative experiments. Consider comparing mechanisms *X* and *Y within* ArchOS. We would have a set of "application" processes (some driver programs to exercise ArchOS) that would have to be changed, if their interface with ArchOS depended on the choice of *X* or *Y*. Ignoring the undesirability of having to develop two sets of driver programs, we would still be faced with the problem of showing (or even believing) that both programs represent comparable stress on the system.

For all these reasons, we will use *only* the Research Requirements to define the clients' view of ArchOS. This (i.e., not using a vocabulary that reflects the inner structure of the OS) will be a novel approach in the specification of OS services. We think such an approach may have value even for a uniprocessor OS, when in a system where all the users are cooperating to meet a common goal. We will define an interface where the client expresses his needs, but not how ArchOS is expected to meet them. We envision requests that supply information like: I need a place to store information, and the value to me of:

- acquiring this place in a time, T_r, is $V_1(T_r)$,
- acquiring a place to store amount, S, of information is $V_2(S)$,
- acquiring a place that has an average access time, T_a, is $V_3(T_a)$,
- and so forth, for things like expected survivability over time, protection from (or accessibility by) others, behavior of the storage place in the event that I (the requestor) crash, behavior of the system in the event that the information is lost (e.g., notify me, kill me), ...

Note that such a storage request could be satisfied with classical *GetMain*, or *AllocateFile*, or *GetMains* (primary and backup, in another failure domain) or *GetMain* with *AllocateFile* (backup), or *AllocateFiles* (primary and backup, in another failure domain).

It is likely that each of the value functions will be accompanied with minimum acceptable and maximum useful values. ArchOS would be expected to satisfy the request, if it can achieve the minimum value for each

function. To the extent that the request can be satisfied in several ways, ArchOS would be expected to maximize the total value achieved without exceeding any function's maximum useful value. One possibility, if the request can't be satisfied, would be to maximize the total value to the system. This might require that resources be taken from another client and used to satisfy this request (note that we assume all clients are cooperating to achieve a single goal, and the value functions they supply should reflect this).

We expect to be able to generalize such a scheme so that it could mimic any strategy, of which we are aware, for handling situations where there are insufficient resources to satisfy all the clients' needs. We are unsure of the degree to which the generalization would make the scheme hazardous. One can easily envision limit cycles, in a heavily loaded situation, where resources are constantly being moved around, and very little use is actually being made of them. It may be necessary to introduce hysteresis by taking resources from another client to supply a new request only if the total system value of all resources increases by more than a certain amount.

To contain the risk inherent in this novel approach, we will allow (as a last resort) the subsequent development steps to restrict the clients from exercising the full generality of the interface.

2.3.3 System Functionality Specification

The System Functionality Specification document will explicitly insist on adherence to the Archons project's guiding concepts. It will formally define certain terms (e.g., atomicity, negotiation, compromise, consensus, "guarantees") that characterize these concepts. It will also specify that certain facilities (e.g., transaction mechanisms, deadlock avoidance/detection mechanisms, recovery mechanisms), representing instances of these concepts, shall exist in ArchOS.

The System Functionality Specification will be derived from the Research Requirements, our previous research, and the assumption that an incarnation of ArchOS resides at each node in a physically dispersed system.

2.3.4 System Architectural Specification

The System Architectural Specification documents a unit of work that takes the Research Requirements, the Clients' Interface Specification and the System Functionality Specification and produces a design of ArchOS, in the form of layered subsystems (e.g., IPC, OS File System, Client File System, OS Resource Allocation, Client Resource Allocation, OS Transaction Server, Client Transaction Server, Timer Services), that satisfy these specifications.

A "uses" hierarchy [Parnas 74] of the subsystems (e.g., IPC "uses" OS Resource Management, and Client

Resource Allocation "uses" IPC) will be produced and justified. Note that a subsystem, *UsingSubsystem*, that "uses" another subsystem, *UsedSubsystem*, can (classically) never make a stronger performance guarantee (with respect to the service supplied by *UsedSubsystem*), than that which is made by *UsedSubsystem*. We intend to examine ways to specify "suspicious use" of another subsystem, *UsedSubsystem*, when we mean that *UsedSubsystem* may be "used", but has probabilistic behavior. In such a way, it may be possible to have *UsingSubsystem* promise more than *UsedSubsystem* does (e.g., by repeated use of *UsedSubsystem*, or confirmation of *UsedSubsystem* through other means).

It is also important to realize that (classically) the proper behavior of *UsedSubsystem* is a precondition for the specified behavior of *UsingSubsystem*. That is, if *UsingSubsystem* can "use" *UsedSubsystem*, then *UsingSubsystem* can exhibit any behavior when *UsedSubsystem* does not perform to specification. While it is certainly true that *UsedSubsystem* can fail in undetectable ways that can only mean that *UsingSubsystem* must fail, it is also true that *UsedSubsystem* can fail in ways that are detectable. Because we are concerned with ultra-reliable systems, we will try to maximize the detectability of the failure of "used" subsystems, and require (where possible) corrective action by the detector (i.e., "user"). Similarly, a precondition of every transaction is the consistency of all the shared data objects that it accesses. A classical transaction can behave in any fashion when this precondition is not met. We would like our transactions to take positive steps, when possible, toward making the data consistent when inconsistencies are detected.

Each subsystem will be defined and will have its behavior specified. Particular concepts and facilities from the Functionality Specification will be associated with appropriate subsystems.

2.3.5 System Design Specification

The System Design Specification document defines and specifies the ArchOS components. A component represents the intersection of an ArchOS subsystem and a hardware node. This is the unit of work that establishes the decentralized nature of ArchOS. This work will be based on the Research Requirements, the System Functionality Specification, and the System Architectural Specification.

It is (at least initially) our intention to have identical node components for any given subsystem. The specification of a component must include the (symmetrical) interfaces with its peers at other nodes. It is through the protocols with its peers that the union of components of a subsystem will supply the services of the subsystem (as required by the System Architectural Specification) using decentralized decision making (concensus, negotiation, and compromise). We will use our interim testbed facility to evaluate and demonstrate the specific decentralized resource management algorithms considered for each ArchOS subsystem.

To minimize initial complexity, we will assume that all interfaces between subsystems occur between the respective components of those subsystems at the same node. That is, all protocols are peer level. After we develop the basic algorithms for a subsystem, and as they are subsequently being refined, we will search for optimizations that may be obtained (and analyze the information hiding that may be lost) by allowing other than peer level protocols. For example, if the OS Resouce Management subsystem component handles a request for disk file space, originating at its node, by asking all its other peer components "how well can you satisfy this request" and analyzing the responses, then it may be possible to have the subsystem that made the request broadcast it to all of these components in the first place.

Wherever possible, the SDS will not make any assumptions about the hardware structure of a node. In the event that it is impractical to specify a component without considering the underlying hardware, we will allow the design to use knowledge of the interim testbed hardware. This point must always be deferred as long as possible and the work that is based on this knowledge must be clearly defined. In this way, a well defined and minimized amount of design must be redone when we move to other hardware.

In a similar fashion, only the communications subsystem will be allowed to have knowledge of the details of the interconnection network(s).

2.3.6 Component Design Specification

The Component Design Specification document defines and specifies the modules that make up each ArchOS component. This work will be based on the Research Requirements, the System Functionality Specification, and System Design Specification. A module represents an ArchOS unit that can be implemented by a single programmer, who is unversed in the Archons project and its goals.

2.3.7 Implementation

The implementation of ArchOS will consist of designing and programming the modules, integrating the modules into components, testing the single node behavior of the component, and (when the communications subsystem components are working) testing the multi-node (i.e., full subsystem) behavior of the component. When all the subsystems have been implemented we will proceed to experiment with ArchOS.

We plan to implement ArchOS from the bottom up (even though we will design it top down), so that testing a component will require test driver programs only to exercise its higher level interfaces.

The module programmers will work from (the appropriate portions of) the ArchOS Component Specification.

2.4 Conclusion

We have arguments and evidence that physically and logically decentralized resource management as we have defined them offer significant potential benefits over conventional approaches in some applications, particularly real-time supervisory control -- e.g., embedded computers for combat platform management and industrial automation.

We are performing conceptual, theoretical, and experimental work to discover the types of benefits which can be achieved, the conditions under which they can and cannot be achieved, and the costs of achieving them.

The Archons project has been progressing for approximately three years, creating and developing the concepts of decentralized resource management, performing theoretical analysis, planning the structures of the operating system and eventual hardware, implementing the interim testbed, etc.

The specification, design, and implementation of ArchOS began this year, and we estimate will require three years for completion of the first experimental prototype. The manpower committed to this ArchOS portion of the Archons project currently consists of five full-time professional position (e.g., post-doctoral) researchers (two of these slots as yet being unfilled), four fulltime Ph.D. students, and part-time participation by several other of the Archons personnel (two faculty, a program manager, and three Ph.D students). Additional staffing will be added as necessary -- for example, programmers when full scale implementation begins.

Further conceptual, formal, and experimental research on the principles, design, and implementation associated with the entire Archons project are continuing concurrently with the ArchOS effort (involving about eight full-time equivalent researchers).

2.5 References

[Ahuja 82] Ahuja, Vijay.
Routing.
In *Design and Analysis of Computer Communication Networks*, chapter 7, pages 233-260. McGraw-Hill, 1982.

[Colwell 83] Colwell, Robert P., Hitchcock, Charles Y., and Jensen, E. Douglas.
A Perspective on the Processor Complexity Controversy.
In *Proceedings, International Conference on Computer Design*. October, 1983.

[DeGroot 74] DeGroot, M.
Reaching a Consensus.
Journal of The American Statistical Association, March, 1974.

[Erman 73] Erman, L. D., Fennell, R. D., Lesser, V. R., and Reddy, D. R.
System Organizations for Speech Understanding.
Proceedings, Third International Joint Conference on Artificial Intelligence, August, 1973.

[Erman 79] Erman, L. D., and V. R. Lesser.
The Hearsay-II System: A Tutorial.
In W. A. Lea (editor), *Trends in Speech Recognition*, chapter 16. Prentice-Hall, 1979.

[Jensen 76] Jensen, E. Douglas.
Distributed Computer Systems.
August, 1976.
Workshop on Distributed Processing, Brown University, Providence, RI.

[Jensen 81a] Jensen, E. Douglas.
Hardware/Software Relationships in Distributed Systems.
In *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 17.
Springer-Verlag, 1981.

[Jensen 81b] Jensen, E. Douglas.
Decentralized Control.
In *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 8.
Springer-Verlag, 1981.

[Jensen 81c] Jensen, E. Douglas.
Decentralized Resource Management.
July, 1981.
Course Notes, C.E.A./I.N.R.I.A/E.D.F Ecole d'ete d'Informatique, Breau-sans-Nappe, France.

[Lynch 83] Lynch, Nancy A.
Concurrency Control for Resilient Nested Transactions.
In *Proceedings, Second SIGACT-SIGMOD Conference on the Principles of Database Systems*. ACM, 1983.

[Marschak 72] Marschak, J., and Radner, R.,
Economic Theory of Teams.
Yale University Press, 1972.

[Parnas 74] Parnas, D. L.
On a 'Buzzword': Hierarchical Structure.
Proc. IFIP 74, 1974.

[Rashid 81] Rashid, Richard F. and George G. Roberston.
Accent: A Communication Oriented Network Operating System Kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Schwarz 82] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, 1982.

[Sha 83] Sha, Lui, E. Douglas Jensen, Richard F. Rashid, and J. Duane Northcutt.
Distributed Cooperating Processes and Transactions.
In *Synchronization, Control, and Communication in Distributed Computing Systems.*
Academic Press, 1983.

[Sha 84] Sha, Lui.
Synchronization in Distributed Operating Systems.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1984.
in preparation.

[Spice 79] Department of Computer Science.
Proposal for A Joint Effort in Personal Computing.
Technical Report, Carnegie-Mellon University, 1979.

3. Transactions for Operating Systems

3.1 Overview

The use of atomic transactions in the kernel of an operating system is one of the fundamentally important aspects of our research. While transactions are well understood at the higher level of database management, most of that knowledge is inapplicable at the lower operating system kernel level. While we believe we were at least one of the first to recognize the value of transactions in operating systems, others have subsequently begun to explore this also. But again our efforts have been pioneering in their focus initially on a formal basis for data consistency and transaction correctness, rather than taking an application-dependent ad hoc approach. Other contributions of the Archons transaction principles are improved modularity and fault tolerance. We base our research on what we term a "relational data model" and "set-wise serializable" atomic transactions.

3.2 Relational Data Model

3.2.1 Introduction

In distributed systems, multiple entities (at any particular level of abstraction) perform tasks by co-operating in various ways so as to improve concurrency, reliability, and modularity, as well as to accommodate physical dispersal. Co-operation implies some form of synchronization among processes or synchronization of concurrent access to shared data objects. The former type of co-operation has been pursued primarily in centralized uniprocessor and multiprocessor computers, while most distributed systems are computer networks and thus focus on the latter type. Furthermore, computer networks (and centralized computers to a lesser extent) typically exhibit a form of co-operation exemplified by autonomous client and server functions. Instead, the Archons project is performing research on the science and engineering necessary for a decentralized computer -- a new hybrid which is a single computer in the sense of a multiprocessor, but is physically dispersed much like a local network. The appropriate paradigm of co-operation in such a machine seems to be peer relationships in which a (variable) number of equal partners collaborate on a function (e.g., to jointly fill a single role). We are particularly interested in styles of co-operation where a team of equals negotiate, compromise, and reach a consensus to manage resources in a global operating system, despite inaccurate and incomplete information within the operating system itself (resulting from communication delays) [Jensen 82]. As a consequence of this situation, the high degree of internal deterministic behavior assumed to be easily achieved in classical centralized computers can be very expensive in distributed systems. Thus, decentralized computers must necessarily be designed to deal with indeterminism explicitly, systematically, and to their best advantage (transparently to their users).

This paradigm has lead us to develop a new relational model of data consistency that allows one to reason about the relationships among collections of processes, data objects, and state variables in distributed systems. Compared with other approaches, such as the conventional serialization model, our model provides greater concurrency in many interesting cases, is free from synchronization-induced deadlock and rollback, and uniformly accommodates both process and data synchronization.

Based on this model, we have created a new approach to distributed co-operating processes, and the concept of co-operating transactions supporting various forms of decentralized control among peers (including both indeterministic and deterministic forms of interaction). Using our model, the synchronization of distributed co-operating processes is formulated as the preservation of a set of dependency relationships among their state variables. In the interest of efficiency, these dependency relationships may be formulated as probabilistic whenever the application permits. Co-operating transactions are co-operating processes whose interactions are made atomic for the sake of reliability. Co-operating transactions cannot be implemented using the conventional serialization model of data consistency because of the generality of the communication involved.

We begin the remainder of this paper by introducing our relational model of data consistency, followed by a description of co-operating processes, and then a discussion of co-operating transactions. Some of these ideas are illustrated by examples from our initial experience in applying them to the Accent [Rashid 81] network operating system and other Spice personal computer system software [Schaffer 82, Ball 81]. These ideas will also appear in the ArchOS operating system for the Archons decentralized computer.

3.2.2 The Relational Model of Data Consistency

3.2.2.1 Our Objections to the Serialization Model

Most of the work on synchronization methods for distributed systems has been done in the context of distributed database systems, and is based on the serialization model of data consistency [Bernstein 80]. The basic concept of the serialization model is that if each transaction executing alone maintains the consistency of the data objects, then executing transactions serially and in any order of execution will also be correct, i.e., maintain the consistency constraints. Therefore, a set of sufficient conditions for the correct concurrent execution of transactions is one which can be proven equivalent to a serial order of execution. One well known form of these conditions is [Papadimitriou 77]:

1. There exists a total ordering of the set of transactions.

2. For every pair of operations that conflict (i.e., at least one operation is a write), their precedence relation on a shared data object must be identical to that of their corresponding transactions in the total ordering of transactions.

Although the serialization model is very general, in the sense that the consistency constraints can be preserved with knowledge of conflicts being the only semantic information about the transactions [Kung 79a], it is inadequate with respect to the needs of distributed operating systems (especially those based on peer relationships rather than client-server type relationships).

- *The serialization model lacks concurrency.* Kung and Papadimitriou [Kung 79a] show that it uses only syntactic (and conflict) information about transactions, and that it is possible to formulate more efficient non-serializable transactions by using information about data objects or additional semantic information about transactions. For example, the work of Lamport [Lamport 76], Kung and Lehman [Kung 79b], Schwarz and Spector [Schwarz 82], Garcia-Molina [Molina 83], and Allchin and McKendry [Allchin 82] all further demonstrate this point. Concurrency is a critical issue in operating systems, and the information needed to improve it is often available (neither of which may be as much the case at the applications level, e.g., in database systems).

- *The serialization model suffers synchronization-induced deadlock and rollback problems* [Bernstein 80]. Synchronization methods based on the serialization model can be classified into two basic approaches -- two phase locking and time stamps. The two phase lock approach can lead to deadlock, while the time stamp approach is prone to problems caused by rollback.

- *The serialization model precludes a distributed (e.g., either decentralized or network) operating system kernel from using atomic transactions for communication and co-operation* [Lamport 76]. When a pair of transactions exchange messages in the course of an interaction, their operations (i.e., the two way communications) might be interleaved so as to violate the relative ordering condition (i.e., 2. above) required by the serialization model.

- *The serialization model does not support the synchronization of co-operating processes.* Co-operating processes must be permitted to change their states autonomously as long as they are not in those states that are governed by the specified rules of co-operation (in our case, the set of dependency relations). However, the serialization model's conditions hold at all times, turning the power of its generality against its use for interprocess co-operation.

To remedy these disabilities, we have supplanted the serialization model with our own model based on relationships among the data objects. We share the premise that each transaction executing alone preserves the consistency constraints of the data objects. But we further assume that the relationships affecting synchronization among the data objects are known. This seems to be a justifiable assumption in our context of distributed operating systems.

In database systems based on the serialization model, serializability is taken as the consistency constraint, i.e., the correctness criterion. In several current efforts on non-serializable transactions, serializability is viewed as a "strong form" of the correctness criteria needed by certain applications and not by others [Schwarz 82, Molina 83, Allchin 82]. In our approach to the correctness issue, consistency constraints are modeled as relations among data objects, and are partitioned into an application independent part called *data invariants* and an application dependent part called *action invariants*. The execution of concurrent

processes or transactions is defined to be correct if it satisfies both the data and action invariants, independent of whether the processes or transactions are serializable. This is because serializability is not a relation among data objects and therefore not a consistency constraint. In our view, serializability is only a set of sufficient conditions to maintain consistency constraints.

3.2.2.2 Classification of Relations

Our relational model of data consistency classifies the possible relationships among data objects as autonomous, dependent, or partially dependent.

- *Autonomous:* The relation is defined as the set of the cartesian products of the domains of the data objects. From a synchronization point of view, the implication of an autonomous relationship is that object A can take on any value that is in its domain, regardless of the value of B (i.e., A and B can be updated separately).

  An autonomous relation will be called probabilistic if a joint probability distribution is defined upon the set of cartesian products. The concept of probabilistic relations is important to our discussion in the section on co-operating processes.

- *Dependent:* The relation is defined by a proper subset of the cartesian products of the domains of the data objects. In this case, the value taken by a data object, A, is constrained by the value taken by another data object, B, and vice versa. The implication of this type relationship is that when there are dependency relationships among data objects, these data objects can no longer be updated independently.

- *Partially dependent:* The relation is defined as a proper subset of the cartesian products of data object domains, a part of which takes the form of cartesian products of subsets of the domains. For example, if the domains of A and B are both {0, 1, 2} with the data invariant "if A=2, then A=B", then the partially dependent relation is the set consisting of the tuple <2,2> concatenated with the set of cartesian products {0, 1} x {0, 1, 2}. The notion of partially dependent relationships allows us to view process synchronization as the act of maintaining the data invariants among distributed state variables. Suppose A and B are state variables of processes P_1 and P_2 respectively. We can interpret the example above as "process P_2 must enter state two if process P_1 enters state two, otherwise processes P_1 and P_2 can change their states autonomously."

3.2.2.3 Definitions

We now proceed to make the following definitions.

- *Data objects:* the user defined, smallest unit of data items that can be synchronized (e.g., locked).

- *Data invariants:* the mathematical representation of the dependency relationships among data objects (e.g., "A=B"). Data invariants must be preserved by all processes or transactions.

- *Atomic data sets:* user defined disjoint sets of data objects, each of which is constrained by a user-specified set of data invariants. For example, one set has data objects A and B with invariant "A=B", and another set has data objects C and D with invariant "C > D". Atomic data sets are our model for the modular decomposition of operating system data objects.

- *Action invariants:* a proper subset of data invariants that are application dependent, i.e., contingent upon the actions of specific transactions. For example, the requirement that the sum of two bank accounts remain unchanged after a fund transfer transaction between them is modeled as the action invariant "either the credit and debit must both be done or neither is done". This requirement must hold at the end of the transaction, but need not hold at other times; i.e., the sum of the two accounts may change across time.

- *Conformity:* a concurrent access to shared data objects which preserves all of the data invariants and satisfies all action invariants. Note that conformal transactions may or may not be serializable.

3.2.2.4 Representations of Data Objects and Data Invariants

Each data object is internally represented by triplets, <name, value, version number>. When a data object is created, its initial value is assigned to version zero of this data object, such as "A[0]: = 1".

When the data object is to be updated, a new version of the object is created and the transaction works on this new version. For example, the code "A: =A+1" in an update transaction corresponds to the following steps:

```
A[v+1] := A[v];          {v is the version number}
A[v+1] := A[v+1] + 1;    {A := A+1}
v := v+1;                {If the transaction commits}
```

If this transaction successfully commits, the new version becomes permanent. Old versions can be kept in the log file as back-ups or discarded. The importance of this representation to us is that it provides a concrete representation of the data invariants. For example, the data invariant "A = B" could be represented as A[v] = B[v], v = 0, 1, 2, 3, ... When a transaction updates the version number of one object in an atomic data set, it then updates the version numbers of all other objects in that set. Since data invariants are defined upon data objects with identical version numbers, a version of an atomic data set exists at a particular time if and only if that version of all its objects exists at that time.

3.2.2.5 Some Important Observations

In this section we state three important observations, based on our model, that are relevant to our later discussion. These observations are presented here in an intuitive fashion, but will appear in a more formal manner in Sha's thesis [Sha 83].

1. *A sufficient condition for conformity:* Conflicting transactions must be mutually exclusive with respect to the version number of the shared atomic data set. That is, data objects belonging to the same version can be shared by several read transactions, but they can only be modified by a single update transaction. Under this condition and our first assumption, a transaction will preserve the

data invariants of each of the accessed atomic data sets. Mutual exclusion with respect to version number can be obtained by using any appropriate synchronization method [Reed 79, Habermann 79, Thomas 79].

2. *Concurrent updating of data objects:* Data objects belonging to different atomic data sets can be updated in any arbitrary order permitted by the action invariants of the updating transaction. This is because there are no data invariants across the boundaries of atomic data sets. Within an atomic data set, a transaction can only update data objects with the same version number (i.e., a transaction can only operate on a particular atomic data set version). However, there can be N concurrent updates on an atomic data set of N data objects. Mutual exclusion with respect to version number means that transactions can concurrently update different data objects of the same atomic data set, as long as these data objects are in different versions. For example, imagine an atomic data set consisting of data objects A and B; transaction T_1 works on A[1] (producing A[2]), and then begins work on B[1]. This would permit a transaction T_2 to begin work on A[2] while T_1 is still working on B[1].

   To support N concurrent updates of an atomic data set with N data objects, two copies for each of the N data objects are needed. One set of copies is used for the atomic data set checkpoint version, while another set is used to store the most recent versions of the data objects. As transactions update the data objects in the atomic data set, the current versions of the data objects could be different. Note that aborting an earlier transaction will lead to the cascaded abortion of later transactions operating on data objects with later version numbers. The trade-off between increased concurrency and the potential for cascaded aborts is an important design issue. Assuming that all the transactions following the checkpoint version are kept in a recovery log, the system can always recover to the checkpoint version after a system failure. A new checkpoint version can be made whenever the current versions of all of the constituent data objects in the atomic data set have the same version number. However, if a fixed interval between checkpoints is desired, then either some concurrency in the updating process must be sacrificed, or additional state save operations will be required. In summary, a small amount of additional storage for the version numbers makes it possible to have both better concurrency and ease of recovery, even when cascaded aborts are involved.

3. *No deadlock or rollback problems result from synchronization.* In our relational model, each of the conflicting transactions will obtain a unique version for each of the atomic data sets accessed by them. Since the data invariants of each of the atomic data sets can be satisfied independent of other atomic data sets, each of these transactions can autonomously produce new versions of the atomic data sets. This cannot cause deadlock, because the generation of new versions makes the atomic data sets available to other transactions. There is also no possibility that this synchronization will produce a rollback, because there are no time stamps used to impose a global order in the execution of transactions.

3.3 A Modular Approach to Non-serializable Concurrency Control: Database Consistency, Transaction Correctness, and Schedule Optimality

3.3.1 Introduction

As part of the Archons decentralized computer system project, we are developing a decentralized operating system with atomic transaction facilities embedded at the kernel level [Jensen 83]. The concurrency control of the executions of transactions has been a very active area of research. A major development in this area is the establishment of the serializability theory [Bernstein 79, Papadimitriou 77]. Since the performance of a distributed computer system depends greatly on concurrency control, the desire to obtain a very high degree of concurrency motivates many to investigate the use of non-serializable schedules.

From a programming point of view, a transaction programmer has two duties. First, he is responsible for the *consistency* and the *correctness* of his transactions. That is, transactions must preserve the consistency of the shared data objects (database) and produce results as specified when executing alone. In the following, we assume that all transactions under discussion are consistent and correct. Second, the programmer must schedule his written transaction according to some scheduling rules implemented by locks or other mechanisms. The concurrency control mechanisms embedded in the transactions allow transactions to be executed concurrently but in such a way that the consistency of the database and the correctness of each transaction are preserved.

Kung and Papadimitriou [Kung 79a] showed that the degree of concurrency provided by any scheduling rule is limited, and the bound is determined by the information used by the scheduling rule. They showed that serializability theory provides the highest possible degree of concurrency, when only information about the classically defined transaction syntax is used. Since serializability theory does not utilize the semantic information of transactions, most of the recent work on non-serializable transactions has focused on the use of semantic information of the transaction system to enhance concurrency [Lamport 76, Schwarz 82, Allchin 82, Lynch 83a, Molina 83]. In this approach, the details of each of the transactions are carefully examined, and a permissible interleaving of transaction steps is then specified accordingly. For example, in [Lynch 83a] transactions are grouped together by some classification scheme. Permissible interleavings for each of the given groups are specified by a corresponding set of "break points" embedded between transaction steps. Sets of break points can be organized into a hierarchical form. Since it is impossible to predict the semantics of various transactions in advance, the transaction system semantic information approach emphasizes the development of syntactic structures to support programmers' specification of their own scheduling rules. The "break point specifications" in [Lynch 83a] and "lock compatibility tables" in [Schwarz 82] are two examples. Schedules consistent with user specifications are defined to be both consistent and correct. It is assumed that programmers understand the details of each others' transactions. They are responsible for the consistency and correctness of their own scheduling rules.

The strength of this transaction system semantic information approach is that it allows programmers to

develop their own concurrency control rules that are tailored to their specific applications. This provides the potential to obtain a very high degree of concurrency. On the other hand, this transaction system semantic information approach does not seem to be suitable for a general transaction facility for two reasons: it neither provides application independent scheduling rules nor addresses the issue of modularity.

An important contribution of the serializability theory is that it provides application independent scheduling rules. As long as programmers follow a prescribed protocol such as the "two phase lock" [Eswaren 76], the consistency and correctness of concurrency control is ensured. It should be noted that serializable schedules allow individual transactions to be regarded as if they are executing alone. The consistency and correctness of serializable schedules follows immediately from the consistency and correctness of individual transactions. When schedules are non-serializable, transactions can no longer be regarded as if they are executing alone. Proving the consistency and correctness of any non-serializable scheduling rule is therefore necessary. Hence, it is very desirable to have a non-serializable concurrency control theory, which provides scheduling rules that are proven to be both consistent and correct. Such rules would release programmers from the burden of inventing their own scheduling rules and then proving their rules to be consistent and correct in each case.

Another difficulty in applying the transaction system semantic information approach to a general transaction facility is that this approach does not address the issue of modularity. Since this is not a common topic in the context of concurrency control, we begin with an example. Consider a database consisting of only two variables A and B with consistency constraint "A + B = 100". Suppose that there are two "fund transfer" transactions: $T_1 = \{t_{1,1}$: A := A - 1; $t_{1,2}$: B := B + 1} and $T_2 = \{t_{2,1}$: B := B - 2; $t_{2,2}$: A := A + 2}, where t_{ij} denotes step j of transaction i. It is easy to verify that, in addition to the serializable schedules, the non-serializable schedule, $\{t_{1,1}; t_{2,1}; t_{2,2}; t_{1,2}\}$, is also consistent and correct. That is, the consistency of the database is preserved and each transaction correctly performs the "fund transfer" task when transactions are executed according to this schedule. Observing this, one might suggest that the transfer transactions be scheduled by means of putting a break point between the two steps in the transaction. Now suppose that we implement transfer transaction T_2 differently by changing the second step "$t_{2,2}$: A := A + 2" to "$t^{\bullet}_{2,2}$: A := 100 - B". When executing alone, the modified T_2, like the original, preserves the consistency of the database and transfers 2 units from B to A. In addition to performing the same function, both versions of T_2 have two steps using the same commutative operators "add" and "subtract". One might suggest putting a break point between step $t_{2,1}$ and $t^{\bullet}_{2,2}$ and scheduling the modified transaction system as before, $\{t_{1,1}; t_{2,1}; t^{\bullet}_{2,2}; t_{1,2}\}$. Unfortunately, this time the schedule always leaves the database inconsistent. For example, let both A and B be 50 initially. Step $t_{1,1}$ changes A from 50 to 49. Step $t_{2,1}$ changes B from 50 to 48. Step $t^{\bullet}_{2,2}$ changes A from 49 to 52. At this point A + B = 100. The last step $t_{1,2}$ adds one to B and leaves the sum of A and B equal to

101. The lesson is that the *specification* of the pre- and post-conditions of transactions is generally insufficient for the specification of break points. To correctly specify permissible interleavings that utilize the semantic information of a group of transactions, programmers must understand the interactions among the steps of all transactions in the group. When the transactions are complex, written and modified by many different programmers from time to time, such a task could quickly become unmanageable. In software engineering, one of the basic principles for the development of a large scale system is to partition it into *implementation independent* modules [Habermann 76]. The interleavings introduced by the transaction system semantic information approach, however, could create implementation sensitive inter-dependence among transactions. As suggested by Molina [Molina 83], it seems appropriate to view the transaction system semantic information approach as a powerful tool to solve specific and static transaction problems that require a very high degree of concurrency, analogous to the VLSI solutions to special computation problems.

Distributed operating systems are known to be very complex, written and modified by many different programmers over a period of years. Any non-modular approach to concurrency control is likely to be unmanageable. Given the difficulty of guaranteeing the consistency and correctness of schedules resulting from applying the transaction system semantic information approach to a general transaction facility, it seems important to develop a new non-serializable approach. This new approach should provide scheduling rules with the following properties:

1. These rules generate only consistent and correct schedules;

2. These rules are modular in the sense that they permit one to write, modify and schedule one's transaction independently, knowing only that other transactions will be consistent, correct and written in the given syntax.

Our approach begins by observing the three types of information available to scheduling rules under the above requirements. The first type is the information about the consistency constraints of the database. Programmers are informed about the consistency constraints, and they are responsible for the preservation of the database consistency. The second type is the syntactic definition of transactions. Transactions must be written in the given syntax. The third type is the semantic information of one's own transaction. In short, we assume that when a programmer is ready to schedule his transaction, he knows the consistency constraints of the database, the details of his transaction which he has just written or modified and the syntax of the transactions that everyone must follow. He makes no assumption about others' transactions except that they are consistent, correct and written in the given syntax.

In this paper, our task is twofold. The first is to develop new syntactic structures that can be used to enhance concurrency. The second is to identify new scheduling rules that make the best use of available information. It turns out that for a given set of primitive steps the best we can do in the development of modular

scheduling rules is to decompose both the database and transactions into *consistency preserving* units of data objects and transaction steps respectively. When transactions and the database are decomposed into such smaller "consistent preserving" units, highly concurrent schedules can then be developed. We would like to mention that these smaller disjoint "consistent preserving" units also facilitate failure recovery, although this topic is outside the scope of this paper and will be pursued elsewhere.

This paper is organized as follows. We first develop the notion of a consistency preserving partition of database. We then develop our model in detail for the classical single level transactions. Next, we extend our results to nested transactions and then to compound transactions. Finally, we investigate the optimality of modular and application independent scheduling rules.

3.3.2 A Model of Operating System Database

There is a general consensus that an operating system should be built in a modular fashion. A typical module, such as the *monitor* [Hoare 74] commonly used in centralized operating systems, consists of a set of shared data objects and a set of pre-defined procedures that facilitate the manipulation of these shared data objects. In addition, there is a simple scheduler embedded in the module to ensure that users access this set of shared data objects in a strictly serial fashion via some mutual exclusion mechanism. However, there is little agreement on what constitutes the basis of a module when shared data objects are distributed across nodes in a distributed computer system. Our approach to this problem focuses on the consistency constraints among system data objects. Given the consistency constraints of the system database, we show that the database can be partitioned into disjoint sets of data objects called *atomic data sets* (ADS). Such a partition is *consistency preserving* in the sense that the consistency of each ADS can be maintained independently, and the conjunction of the consistency constraints of atomic data sets is equivalent to the consistency constraints of the entire database. It will also be shown that there always exists an unique maximal consistency preserving partition.

From an application point of view, atomic data sets can be used as a basis for constructing distributed software modules: modules that encapsulate distributed data objects. For example, one can easily generalize the monitor (or abstract data type) approach developed for centralized systems as follows. We can define a set of primitive procedures for each of the data objects in an atomic data set to facilitate the manipulation of these data objects. When the scheduling rules grant one the privilege, one is entitled to use those pre-defined procedures as building blocks of his own transaction. Before the development of the formalism, we would, however, like to first present an example to illustrate the concepts of the consistency preserving partition of the database. We also use this example to provide an intuitive discussion of some issues related to the design of consistency constraints of a distributed system. The investigation of the principles of designing consistency constraints for a distributed operating system to enhance system performance is likely to become an important area of research.

3.3.2.1 Consistency Preserving Partition of Database --- An Example

The very nature of a distributed system provides us with both the opportunity of realizing a very high degree of parallelism and the difficulty of coping with large communication delays. In order to maximize the benefit of parallelism and to minimize the the performance penalty caused by communication delay, it is often useful to consider the use of consistency constraints that are weaker than the corresponding ones in centralized systems. We illustrate this idea by a simplified case of managing the directories of a file system.

We consider a set of shared system files distributed at different nodes. There is a local directory (LD) at each node indicating the resident files. With only these local directories, one must potentially search through all the LD's in order to locate a file, and this would be very inefficient. To increase efficiency, the system has a global directory (GD). The GD indicates which LD should be searched for each of the shared files. The GD is replicated for reliability and performance. When one needs a file, the local operating system kernel will first search through its LD, and then it will search a nearby GD, if the file is not in its LD. The introduction of GD's facilitates file look-ups, but in a large system the GD's can become a performance bottle-neck. To further improve the efficiency, the local operating system kernel at each node constructs a partial global directory (PGD) which indicates the resident nodes of the frequently used remote files.

Although GD's and PGD's help in locating files, they also make the updating process more complicated. One could define the set of consistency constraints of all the GD's, PGD's and LD's as the requirement that they must always point to the correct locations with respect to any reference. This implies that when one moves a file from one machine to another, the updating of the source and destination LD's, the GD's and all the relevant PGD's must appear as an instantaneous event with respect to other transactions. This can be accomplished by following the two phase lock protocol [Eswaren 76] to lock the source and destination LD's, all the GD's and all PGD's that contain an entry indicating the transferred file. However, this approach has a serious drawback in performance, because the two phase lock requires that no lock can be released until all the locks have been obtained. In short, the entire system's file look-up activities might be forced to or near a halt by a few file transfer operations. Therefore, it seems reasonable to seek an alternative approach. One simple alternative which permits a higher degree of concurrency is to use "recent" historical locations in lieu of the current ones. Such a tactic is quite common in distributed systems and is modelled as follows. First, GD's and PGD's can point to any valid LD location, i.e. the relationship among GD's, PGD's and LD's is in the form of Cartesian products. Second, we have the following two performance enhancement schemes. First, GD will be updated whenever a LD is updated. Second, PGD's are managed by a "fault driven" policy. When a transaction uses a PGD, it will increment the "success-counter" or the "failure-counter" associated with the PGD according to the result from using its information. The local operating system kernel will periodically compute the percentage of reference failures. Should it exceed a threshold, the entries in the

PGD will be updated by using information in the GD's. Obsolete forwarding addresses will also be deleted in this updating process. The simplest scheme for a transaction directed to the "wrong" LD is to abort and try later. There are more sophisticated schemes which can enhance performance. For example, one is to require the transfer transaction to leave a forwarding address in the PGD of the source node. From a programming point of view, performance enhancement schemes are transaction specifications that can be implemented by *asynchronous* processes. Conceptually, consistency constraints define the set of consistent states. Nevertheless, from an application point of view certain consistent states are considered to be more favorable than others. Performance enhancement schemes are designed to increase the *probability* of staying in the most favorable consistent states.

Generally speaking, weaker consistency constraints permit a higher degree of concurrency. However, once the consistency constraints are weakened, the complexity of transactions will be increased for two reasons. First, the process of weakening the consistency constraints enlarges the number of system states that are considered to be consistent. For example, if the set of consistency constraints regarding GD and PGD is relaxed, a transaction must be written to function correctly in the case that GD or PGD will only give a *valid* LD location but not necessarily the LD location where the file actually resides. That is, a transaction must be able to abort when the file cannot be found or traced. Transactions must have the ability to deal with all the possible system states that are consistent. Second, strong consistency constraints generally ensure that the system will stay in a small set of favorable states, although enforcement could be too expensive. When the set of permissible state is enlarged, transactions must generally redesigned to better keep the system in favorable states. This also increases the complexity of transactions. The evaluation of the trade-offs between system concurrency and transaction complexity is an exciting new research area. However, in this paper we will not analyze the performance trade-offs, but rather focus on the notions of database consistency, transaction correctness, and schedule optimality.

3.3.2.2 Data Objects, Database and Consistency

A *data object*, O, is a user defined smallest unit of data which is individually accessible and upon which synchronization can be performed (e.g. locking). Associated with each data object O, we have a set Dom(O), the domain of O, consisting of all possible values taken by O. The granularity of a data object is not important to the discussion of consistency and correctness. For example, a local directory can be designated as a data object. Alternatively, each entry in this directory can be designated to be a data object, and the directory can be considered to be a collection of data objects so as to permit concurrent operations on the directory.

Each data object is internally represented by triplets, <name, value, version number>. When a data object is created, its initial value is assigned to version zero of this data object, e.g. "A[0]: = 1". When the data object is updated, a new version of the object is created and the transaction works on this new version. The version

number will be incremented when the update is completed. For example, the step "Λ := Λ+1" in an transaction corresponds to the following:

```
A[v+1] := A[v];   {where v denotes the current version}
A[v+1] := A[v+1] + 1;
v  := v+1;
```

In the following discussion, when we just refer to the current value of a data object O, we would write "O" instead of "O[v]". The version number representation will be used only if different versions of the values of a data object are referred to.

The system database $D = \{O_1, O_2, ..., O_n\}$ is the collection of all the shared data objects in the system. A state of D is an n-tuple $Y \in \Omega = \prod_{j=1}^{n} Dom(O_j)$. Associated with D, there is a set of consistency constraints in the form of predicates on the states of D. A consistent state of D is an n-tuple, Y, satisfying this set of consistency constraints. This is indicated by "C(Y) = 1", where C is a Boolean function indicating if this set of consistency constraints is satisfied by Y. For simplicity, we will also refer to this set of consistency constraints by C. The meaning of C is easily determined by the context. The set of all consistent states of D is denoted by U, where $U = \{Y \mid C(Y) = 1\}$.

To illustrate these ideas, we return to the example of Section 3.2.2.1. Suppose that our operating database consists of only these directory objects GD's, PGD's and LD's. A state of the database forms a description of the locations of system files.

3.3.2.3 Consistency Preserving Partition of Database

Once we decide upon the set of consistency constraints, we want to determine the concurrency permitted. The concurrency permitted by a given set of consistency constraints is determined by partitioning the operating system database into disjoint sets called *atomic data sets* (ADS), whose consistency can be maintained independent of each other. Each atomic data set has its own consistency constraints, and the conjunction of all the ADS consistency constraints is equivalent to the consistency constraints of the database. In the following, we first develop the notion of a consistency preserving partition and show that there is always a unique maximal partition with respect to a given set of consistency constraints.

Let $I = \{1, 2, ..., n\}$ be the index set of D. The index $i \in I$ specifies the data object $O_i \in D$. Let $\pi_S(Y)$ denote the projection of an n-tuple $Y \in \Omega$ using the set of indices $S \subset I$. That is, $\pi_S(Y)$ denotes the tuple whose elements are the values of the data objects indexed by S. Let $P = \{S_1, ..., S_k\}$ denote a partition of I.

Let V_i be the set whose elements are the projections of all the consistent states, $X \in U$, onto an arbitrary index set S_i, i.e. $V_i = \bigcup_{X \in U} \{\pi_{S_i}(X)\}$.

Definition: A partition of the index set I, $P = \{S_1, S_2, \ldots, S_k\}$, is said to be consistency preserving (CP) if and only if,

$$\forall (Y \in \Omega)\{ [\pi_{S_i}(Y) \in V_i, i = 1 \text{ to } k] \rightarrow [Y \in U] \}.$$

An atomic data set $\mathcal{A}_i$ for a CP partition P is the set of data objects specified by $S_i \in P$. The associated partition of data objects in D, Q, is called a consistency preserving partition of D.

The definition of a CP partition states that a CP partition has the property that any choice of the consistent states of the atomic data sets leads to a consistent state of the database. The following theorem shows that the consistency constraints of the database can be decomposed into sets of ADS consistency constraints.

Let P be a CP partition of I and let $S_i \subset I$ be the set of indices which specify the data objects in the atomic data set $\mathcal{A}_i \subset D$. Let the set of all the consistent states of an ADS $\mathcal{A}_i$ be $U_i = \bigcup_{X \in U}\{\pi_{S_i}(X)\}$.

Definition: The set of consistency constraints C_i whose truth set is the consistent states of $\mathcal{A}_i$ is called the ADS consistency constraints of $\mathcal{A}_i$, i.e.

$$U_i = \{ \pi_{S_i}(Y) \mid C_i(\pi_{S_i}(Y)) = 1 \}$$

Theorem 1: The conjunction of all the ADS constraints C_i, $i = 1$ to k, is equivalent to consistency constraints C of D. That is,

$$C = C_1 \wedge C_2 \wedge \ldots C_k.$$

Proof: Let U^* be the truth set of the conjunction of all the ADS constraints. We have,

$$U^* = \{Y \mid C_i(\pi_{S_i}(Y)), i = 1 \text{ to } k\}$$

$$= \{Y \mid \pi_{S_i}(Y) \in U_i, i = 1 \text{ to } k\} = U.$$

Hence, $C = C_1 \wedge C_2 \wedge \ldots C_k$. □

CP partitions exist, since the trivial partition { I } is CP. Furthermore, the CP partitions are partially ordered by refinement. That is, for any CP partition P_1, P_2; P_1 is refined by P_2 if and only if

$\forall(S_j \in P_2)\ \exists(S_i \in P_1)(S_j \subset S_i)$. A maximal CP partition is one which is refined by no other CP partition. In the following, we prove that there exists a unique maximal CP partition P.

The proof is based on three Lemmas. The idea of Lemma 2-1 is illustrated by the following example. Let $P_1 = \{ \{1, 2\}, \{3\} \}$ be a CP partition of the index set $I = \{1, 2, 3\}$. Suppose that $A = (a_1, a_2, a_3)$ and $B = (b_1, b_2, b_3)$ are two consistent states. Let S be a partition set, either $\{1, 2\}$ or $\{3\}$. Lemma 2-1 states that the two new states which result from swapping the projections of A and B specified by S are also consistent. That is, (a_1, a_2, b_3) and (b_1, b_2, a_3) are consistent.

Define a mapping $H_S: \Omega \times \Omega \rightarrow \Omega$ as follows, where $S \subset I$. Given that $X_1, X_2 \in \Omega$, $H_S(X_1, X_2) = Y$, where Y satisfies $\pi_S(Y) = \pi_S(X_2)$ and $\pi_{S^c}(Y) = \pi_{S^c}(X_1)$, where $S^c = I - S$. Thus $H_S(X_1, X_2)$ replaces the projections of X_1 specified by S with the projections of X_2 specified by S.

Lemma 2-1: Suppose that $X_1, X_2 \in U$. If S is an element of any CP partition of I, then $H_S: U \times U \rightarrow U$.

Proof: If $S = I$, then $H_S(X_1, X_2) = X_2$, and the result follows.

Let $P = \{S, \sigma_1, ..., \sigma_k\}$, $k \geq 1$ be a CP partition.

Define $W_0 = X_2$, $W_i = X_1$, $i = 1$ to k; so that $W_i \in U$, $i = 0$ to k.

$\pi_S(W_0) = \pi_S(H_S(X_1, X_2))$

$\pi_{\sigma_i}(W_i) = \pi_{\sigma_i}(H_S(X_1, X_2))$, $i = 1$ to k.

Given that P is CP, $H_S(X_1, X_2)$ is therefore in U by the definition of a CP partition. Thus H_S maps pairs of consistent state into a consistent state. □

When we have two or more distinct CP partitions of the same index set I, partition sets from distinct partitions could intersect. Lemma 2-2 generalizes Lemma 2-1 by allowing the intersections to be used for the specification of swapping. For example, let $P_2 = \{ \{1\}, \{2, 3\} \}$ be a second CP partition. The intersection of $\{1, 2\} \in P_1$ and $\{2, 3\} \in P_2$ is $\{2\}$. Lemma 2-2 states that the two states resulted from swapping the projections of A and B specified by $\{2\}$ are also consistent. That is, (a_1, b_2, a_3) and (b_1, a_2, b_3) are consistent. We show the consistency of (a_1, b_2, a_3) as follows. First, we use Lemma 2-1 to swap the projections of A and B specified by $\{1\}$ of P_2. $E = (a_1, b_2, b_3)$ is one of the two resulting consistent states. Next, swapping the projections of A and E specified by $\{3\}$ of P_2, we find (a_1, b_2, a_3) to also be a consistent state. We now give a general proof of Lemma 2-2.

Lemma 2-2: Suppose that $S \in P_1$ and $\sigma \in P_2$, where P_1 and P_2 are CP. Then $H_{S \cap \sigma}: U \times U \rightarrow U$.

Proof: If $S \in P_1$, then there exists a CP partition P such that $S^c \in P$. Since $X_1, X_2 \in U$, it follows that $H_{S^c}(X_1, X_2) \in U$ by Lemma 2-1. Therefore, $H_\sigma(X_1, H_{S^c}(X_2, X_1)) \in U$ as well. The Lemma follows, since $H_\sigma(X_1, H_{S^c}(X_2, X_1)) = H_{S \cap \sigma}(X_1, X_2)$. □

Lemma 2-3 demonstrates that the least common refinement of any two CP partition is also a CP partition. For example, the least common refinement of P_1 and P_2, {{1}, {2}, {3}}, is also CP. In this case, given a set of consistent states such as (a_1, a_2, a_3), (b_1, b_2, b_3) and (c_1, c_2, c_3), we must prove that $\{a_1, b_2, c_3\}$ is also consistent. First, we apply the intersection of {1} and {1, 2} to "A, B" and "A, C" respectively. (a_1, b_2, b_3) and (a_1, c_2, c_3) are two of the four new consistent states. Next, we apply the intersection of {2, 3} and {3} to these two new states. One of the two resulting consistent states is $\{a_1, b_2, c_3\}$. We now give a general proof of Lemma 2-3.

Lemma 2-3 if P_1 and P_2 are CP, then their least common refinement is also CP.

Proof: Let $P_1 = \{S_1, \ldots, S_m\}$ and $P_2 = \{\sigma_1, \ldots, \sigma_n\}$. Their least common refinement is $P_1 \cap P_2 = \{C_1, \ldots, C_L\}$, where $C_i = S_j \cap \sigma_k$ for some j, k, i = 1 to L.

Let $X_i \in U$, i = 1 to L and $Y \in \Omega$ be given such that $\pi_{C_i}(X_i) = \pi_{C_i}(Y)$, i = 1 to L. We must prove $Y \in U$ to conclude that the $P_1 \cap P_2$ is CP.

Define a sequence $\{X^*_i, i = 1 \text{ to } L\}$ as follows: $X_1^* = X_1$, $X_j^* = H_{C_j}(X^*_{j-1}, X_j)$, j = 2 to L. Noting that $C_j = S_i \cap \sigma_k$, Lemma 2-2 indicates that $X_j^* \in U$, j = 1 to L. It follows that $X^*_L = Y \in U$. □

Theorem 2: There exists a unique maximal CP partition.

Proof: Suppose that there exists more than one maximal CP partition. The least common refinement of distinct maximal CP partitions is CP by Lemma 2-3, thus contradicting the maximality assumption. □

Corollary 2: There exists a unique maximum CP partition Q of D.

Theorem 2 indicates that there is a CP partition that is "most refined" with respect to a given set of consistency constraints. This partition will allow the maximal concurrency of transactions, although any CP partition can be used. In the directory example discussed earlier, a consistency preserving partition of the directories could be as follows:

- Each GD is an atomic data set, with the ADS consistency constraint that each entry points to a valid node location.

- Each PGD is an atomic data set with the ADS consistency constraint that each entry points to a valid node location.

- All the LD's are placed in one single atomic data set with the consistency constraints that any file name appears in one and only one LD where the file resides.

Note that in this formulation both PGD's and GD's are only required to point to any valid node location. From a performance enhancement point of view, they are treated differently. GD's are required to be updated whenever a file is moved, whereas a PGD is updated only when the percentile of reference failures exceeds a threshold. Note that the required frequencies for updating GD's and PGD's respectively are performance enhancement schemes designed to help GD's and PGD's in pointing to relatively "recent historical locations". Scheduling rules will *not* take performance enhancement schemes into consideration. Conceptually, system consistency constraints are the laws defining the legal states of the system database. Scheduling rules ensure that these laws are observed by the concurrent executions. Performance enhancement requirements are used to help the system to stay in the most favorable legal states. The implementation of performance enhancement requirements is in the form of asynchronous processes and not a part of concurrency control.

In the following section, we will develop the notion of transaction systems and show how consistency preserving partitions can be used to schedule transactions in a non-serializable fashion.

3.3.3 A Model for Transaction Systems

A transaction system is a set of transactions that share a common database. Given a transaction system, a modular scheduling rule independently partitions the steps of each transaction into equivalent classes called *atomic step segments*. Having partitioned a transaction, one can use "locks", "time stamps" or other protocols to ensure that the atomic step segments specified by the rule will be executed serializably. For example, the serializability theory is a special modular scheduling rule which considers each transaction in the system to be a single atomic step segment. A formal model of modular scheduling rules and their properties will be presented in section 3.2.4, in which we show that the setwise serializable scheduling rule is optimal in the set of all the application independent scheduling rules and generalized setwise serializable scheduling rules form a complete class within the set of all the modular scheduling rules. In this section, we focus on the *consistency and correctness of the schedules* associated with these two rules.

A schedule z is said to be consistent if the execution of the transaction system according to z preserves the consistency of the database. z is also said to be correct if the execution leads to the satisfaction of the post-condition of each of the transactions. It is important to point out that the concept of correctness applies only to the relationship between the inputs and outputs of each individual transaction, not the aggregate effect

of executing a set of transactions. The aggregate effect is dealt with through the notion of database consistency. Suppose that we have a transaction withdrawing $5.00, a transaction depositing $10.00 and an account with current balance equal to zero. In addition, let the non-negativity of the account balance be the consistency constraint. The withdrawal transaction aborts if it encounters a balance less than $5.00. Depending upon the order of execution, the withdrawal could be either successful or "bounced". We consider a schedule z for these two transactions to be correct if under z each of the two transactions does what it is supposed to do, independent of whether the withdrawal is successful or is "bounced". We consider the schedule to be consistent if at the end of execution the account balance is non-negative.

This section is organized as follows. We first study transaction systems composed of single level transactions. We define the notion of setwise serializable schedules and prove their consistency and correctness. Next, extend our work to nested transactions. We then introduce a new transaction syntax called a *compound transaction* and define the associated schedules called *generalized setwise serializable schedules.* We conclude this section by proving the consistency and correctness of generalized setwise serializable schedules. Finally, we want to point out that throughout this section, transactions step are classified into "read" and "write". The possible use of a richer set of primitive steps will be discussed in Section 3.2.4.2.

3.3.3.1 Single Level Transactions

The study of transaction systems composed of single level transactions forms the basis for our later work on nested transactions and compound transactions. This section is organized as follows. First, we define the syntax of single transactions. Second, we define the notions of schedules, equivalent schedules and setwise serializable schedules. Third, we define the notion of consistency and correctness and prove that setwise serializable schedules are consistent and correct. We conclude this section by developing algorithms for the enforcement of setwise serializability.

3.3.3.1.1 Syntax

In this section, we define the syntax of single level transactions and the notions of pre- and post-conditions of a transaction. We first define the syntax of single level transactions.

Definition: A single level transaction T_i is a sequence of transaction steps $(t_{i,1}, t_{i,2}, ..., t_{i,m_i})$ with the following syntax:

<SingleLevelTransaction> ::= BeginTransaction <StepList> EndTransaction.

<StepList> ::= <Step> | <StepList>;<StepList>;

<Step> ::= ReadStep | WriteStep

A transaction step, either "read" or "write", is modelled as the non-divisible execution of the following instructions [Kung 79a]:

$$L_{t_{i,j}} := O_{t_{i,j}}$$

$$O_{t_{i,j}} := f_{t_{i,j}}(L_{t_{i,1}}, L_{t_{i,2}}, ..., L_{t_{i,j}})$$

where $t_{i,j}$ represents step j of transaction T_i; $L_{t_{i,j}}$ is the local variable used by $t_{i,j}$ to store the value read. $O_{t_{i,j}}$ is a data object accessed by $t_{i,j}$ and $f_{t_{i,j}}$ represents the computation. In this model, every step reads and then writes a data object. A read step is interpreted as writing the value read back to the data object, i.e. the $f_{t_{i,j}}$ associated with a read step is the identity function.

We now define the notions of input steps, output steps, pre-condition and post-condition of a transaction.

Definition: Let $T_i = \{t_{i,1}, ..., t_{i,m_i}\}$ be a transaction. Let the data object O be the one accessed (read or written) by step $t_{i,j}$. Step $t_{i,j}$ is said to be an *input step* if it is the step in T_i that first accesses data object O. Step $t_{i,j}$ is said to be an *output step* if it is the step in T_i last accessing O. That is, for every data object O accessed by T_i there are an input step and an output step associated with O. Note that when there is only one step in T_i accessing O, then this step is both an input and an output step.

Definition: Let $O_{m,j}[v_i]$, j = 1 to k, be the set of values read by the input steps of T_m, where v_i denotes the version of a data object that is input to a transaction. Let the index set of $O_{m,j}[v_i]$, j = 1 to k, be I_m. The input values to T_m, $O_{m,j}[v_i]$, j = 1 to k, are said to satisfy the pre-condition of T_m, if and only if

$$\exists (X \in U)(\pi_{I_m}(X) = O_{m,j}[v_i], j = 1 \text{ to } k.)$$

That is, a transaction must function properly if all the values input to the transaction could have come from a consistent state of the database.

Definition: Let $O_{m,j}[v_f]$, j = 1 to k, be the set of values written by the output steps of transaction T_m, where v_f denotes the version of a data object output by the transaction. The *post-condition* of transaction T_m is the specification of the output values of T_m as functions of the input values,

$$O_{m,j}[v_f] = g_{m,j}(O_{m,1}[v_i], ..., O_{m,k}[v_i]), j = 1 \text{ to } k.$$

3.3.3.1.2 Schedules and Setwise Serializability

Given a consistency preserving partition of the database, the setwise serializable scheduling rule partitions each transaction into a special form of atomic step segments called *transaction ADS segments*. A transaction ADS segment of a transaction T_i is simply all the steps in T_i that access the same ADS. A schedule z is said to be setwise serializable if all the transaction ADS segments in the system are executed serializably under z. The purpose of this section is to formally define the notion of setwise serializability and to identify the conditions under which a schedule is setwise serializable.

Definition: A transaction ADS segment is the sequence of steps in a transaction that accesses the same ADS. Let $\Psi(i, \mathcal{A})$ denote the transaction ADS segment of transaction T_i accessing ADS $\mathcal{A}$. Let $t_{i,j} > t_{k,m}$ denote that step $t_{i,j}$ is executed after step $t_{k,m}$.

1. $\Psi(i, \mathcal{A}) = \{t \mid (t \in T_i) \wedge (t \text{ reads or writes a data object in ADS } \mathcal{A})\}$

2. $\forall((t_{i,j}, t_{i,k} \in \Psi(i, \mathcal{A})) \wedge (t_{i,j} > t_{i,k}))((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,j} > t_{i,k}))$

We now define the notions of transaction systems and their schedules. Next, we define the notion of a setwise serial schedule.

Definition: A transaction system T is a finite set of transactions $\{T_1, ..., T_n\}$ operating upon the shared database D.

Definition: A schedule z for transaction system T is a totally ordered set of all the steps in the transaction system $T = \{T_1, ..., T_n\}$ such that the ordering of steps of T_i, i = 1 to n, in the schedule is consistent with the ordering of steps in the transaction T_i, i = 1 to n.

$$[\forall(t)((t \in z) \Leftrightarrow (t \in T))] \wedge [\forall(T_i \in T)\forall((t_{i,j}\ t_{i,k} \in T_i) \wedge (t_{i,k} > t_{i,j}))((t_{i,j}, t_{i,k} \in z) \wedge (t_{i,k} > t_{i,j}))]$$

Definition: A setwise serial schedule is a schedule in which transaction ADS segments accessing the same ADS do not overlap. Let $t_i^{A,1}$ and $t_i^{A,m}$ denote the first step and the last step of transaction ADS segment $\Psi(i, \mathcal{A})$ respectively. Let Q be a consistency preserving partition of D. A schedule z for transaction system T is said to be setwise serial if and only if under z,

$$\forall(\mathcal{A} \in Q)\forall(T_i \in T)\forall(t^A \in z \wedge t^A \notin T_i)((t_i^{A,1} > t^A) \vee (t^A > t_i^{A,m}))$$

where t^A represents any step accessing ADS $\mathcal{A}$ in the transaction system T.

Having defined the notion of setwise serial schedules, we want to define a setwise serializable schedule as

one which is computationally equivalent to a setwise serial schedule. This requires us to first define the notion of equivalent schedules.

Definition: Let $O_{t_{i,j}} \equiv O_{t_{k,m}}$ denote step $t_{i,j}$ and step $t_{k,m}$ read or write the same data object. A schedule z for transaction system T is said to be *equivalent* to another schedule z^* for T, if for every pair of steps $t_{i,j}$ and $t_{k,m}$ in z and z^*,

$$\forall ((t_{i,j}, t_{k,m} \in T) \wedge (O_{t_{i,j}} \equiv O_{t_{k,m}})) [((t_{i,j}, t_{k,m} \in z) \wedge (t_{i,j} > t_{k,m})) \Leftrightarrow ((t_{i,j}, t_{k,m} \in z^*) \wedge (t_{i,j} > t_{k,m}))]$$

That is, the partial orderings of steps on each of the shared data objects in z and z^* are identical. In addition, the orderings of steps in both z and z^* are also consistent with the internal orderings of steps in each of the transactions in T, since z and z^* are schedules for the same transaction system T. Hence, for any given initial state of D the executions of T according to z and z^* give the same computation in the sense that they yield the same sequences of values for each data object in the database and the same sequences of values for each of the local variables (states) of each transaction in T.

Definition: A schedule z for transaction system T is said to be setwise serializable if there exists a setwise serial schedule z^* for T such that z and z^* are equivalent.

Setwise serializability is determined by the transaction ADS segment precedence graph. In the following, we first define the notion of a general precedence graph.

Definition: Let z be a schedule for transaction system T. Let τ be a sequence of transaction steps in T. Γ be a finite set of τ. Let Σ be a set of data objects in D. A *precedence graph* $G(z, \Gamma, \Sigma)$ is a directed graph whose nodes are elements of Γ. An arc $\langle \tau_i, \tau_j \rangle$, which represents that τ_i preceeds τ_j, exists if the execution of T according to z results in one of following three conditions:

1. there exists a data object $O \in \Sigma$ for which τ_i reads from O *immediately* before τ_j writes into O;
2. there exists a data object $O \in \Sigma$ for which τ_i writes into O *immediately* before τ_j reads from O;
3. there exists a data object $O \in \Sigma$ for which τ_i writes into O *immediately* before τ_j writes into O.

As an example, the familiar transaction system precedence graph for schedule z is represented by G(z, T, D), where z is a schedule for transaction system T, and D is the database.

Definition: Let $\Xi_{ADS}(T_i)$ be the partition of the steps of transaction T_i such that each element of the partition is a transaction ADS segment. Let Γ be the set of all the transaction ADS segments in transaction system T, i.e. $\Gamma = \{\Psi \mid \Psi \in \Xi_{ADS}(T_i) \wedge T_i \in T\}$. The precedence graph $G(z, \Gamma, D)$ is called the *transaction ADS segment precedence graph* for schedule z.

Let "Cycle(G) = 0" denote that the graph G contains no cycle.

We now prove that a schedule z for a transaction system T is setwise serializable if the transaction ADS segment precedence graph for z contains no cycle. We must show that there always exists a setwise serial schedule z^* in which the partial ordering of steps on each of the data objects is the same as that in z. To demonstrate this, we use the procedure known as *topological sorting* [Aho 83]. Topological sorting creates a total ordering that is consistent with all the partial orderings represented by a directed acyclic graph. We first use this procedure on the transaction ADS segment graph to create a list of *partial setwise serial* schedules, each of which is a serial schedule for all the transaction ADS segments accessing the same ADS. Note that the transaction ADS segment graph does not consider the step orderings between different transaction ADS segments defined by the individual transactions. For example, a transaction T_i with four steps can have its 1st and 3rd steps accessing ADS $\mathcal{A}_1$ while the 2nd and 4th steps accessing ADS $\mathcal{A}_2$. That is, $\{t_{i,1}, t_{i,3}\}$ and $\{t_{i,2}, t_{i,4}\}$ are the two transaction ADS segments of T_i. The precedence relation from step 1 to 2, 2 to 3 and 3 to 4 are not considered by the transaction ADS segment graph. To create the setwise serial schedule, we must take these internal step orderings into account. Therefore, we now create a *transaction system step precedence graph*. Each node in the graph is a step in T. We first draw arcs to represent all internal orderings between steps in each of the transactions. An arc is drawn from node i to node j if step i immediately preceeds step j in the same transaction. Next, we draw arcs to represent the partial orderings that are defined by the partial setwise serial schedules. An arc is drawn form node k to node m if step k immediately preceeds step m in one of the partial setwise serial schedules. Once this is done, we use the topological sorting procedure to create a total ordering which then gives the required setwise serial schedule.

Theorem 3: A schedule z for transaction system T is setwise serializable if there is no cycle in the transaction ADS segment precedence graph for z, i.e. Cycle($G(z, \Gamma, D) = 0$).

Proof: We first use the topological sorting procedure on the transaction ADS segment graph to create a list of partial setwise serial schedules. There must be a node in the transaction ADS segment precedence graph for z that has no entering arcs. Otherwise, there is a cycle in the graph. Suppose that this node corresponds to $\Psi(i, \mathcal{A})$. List transaction ADS segment $\Psi(i, \mathcal{A})$ on the partial setwise serial schedule z_A for ADS $\mathcal{A}$. Remove $\Psi(i, \mathcal{A})$ from $G(z, \Gamma, D)$ and repeat the procedure until all the nodes are removed from $G(z, \Gamma, D)$. We now create the transaction system step precedence graph in which each node represents a step in T. We draw an arc from node i to node j if step i immediately preceeds step j in the same transaction. We also draw an arc from node k to node m if step k immediately preceeds step m in one of the partial setwise serial schedules. Having completed the graph, we perform the topological sorting procedure on the graph. The resulting total ordering of steps is a setwise serial schedule z^*. The total ordering of steps in z^* is consistent with the internal orderings of steps defined by the transactions in T and is consistent with the partial orderings of steps on each of the data objects in $G(z, \Gamma, D)$. □

It is worthwhile to point out that setwise serializable schedules do not generally prohibit cycles from being formed in the transaction system precedence graph, they only prohibit cycles from being formed in the transaction ADS segment precedence graph. Setwise serializability reduces to serializability if the database consists of one ADS.

3.3.3.1.3 Consistency and Correctness

When serializability is used as the criterion of correctness for concurrency control, the notions of data consistency and transaction correctness follow directly from the assumptions that each transaction terminates, preserves the consistency of the database and produces correct results when executing alone. When schedules are non-serializable, transactions can no longer be regarded as if they are executing alone. Therefore, we must prove the consistency and correctness of any non-serializable schedule. We consider a schedule to be consistent and correct, if the execution of the transaction steps according to the schedule preserves the consistency of the database and satisfies the post-condition of each of the transactions. Our fundamental assumptions about a transaction are as follows:

- A1 *Termination*: A transaction is assumed to terminate.
- A2 *Transaction Correctness*: A transaction is assumed to produce results that satisfy its post-condition when executing alone and when the database is initially consistent.
- A3 *Data Consistency*: A transaction is assumed to preserve the consistency of the database when executing alone.

Definition: A transaction T_i is said to be consistent and correct if and only if T_i satisfies assumptions A1, A2 and A3. A transaction system T is said to be consistent and correct if and only if all the transactions in T are consistent and correct.

Definition: A schedule z for transaction system T is said to be consistent if and only of the execution of T according to z preserves the consistency of the database D.

Definition: A schedule z for transaction system T is said to be correct if and only if the execution of T according to z satisfies the post-condition of each of the transactions in T.

Before proving that setwise serializable schedules are both consistent and correct, we need to define the notion of equivalent executions of a given transaction under different schedules.

Definition: Let z and z^* be two schedules for transaction system T. Let t be a step of transaction T_i in T. Let the values of the data object accessed by t in z and z^* be O and O^* respectively. Let the values of the local variable associated with t in z and z^* be L and L^* respectively. Transaction T_i is said to be executed equivalently under z and z^* if and only if

$$\forall (t \in T_i)((O = O^*) \wedge (L = L^*))$$

Theorem 4: If T_i is executed equivalently under two different schedules z and z^* then the post-condition of T_i will either be satisfied under both z and z^* or not satisfied under both z and z^*.

Proof: Let the values input to T_i be $O_{i,1}[v_i]$, ..., $O_{i,k}[v_i]$. Let the values output by T_i be $O_{i,1}[v_f]$, ..., $O_{i,k}[v_f]$. The post-condition of T_i is the specification of the output values of T_i as some functions of the input values: $O_{i,j}[v_f] = f_j(O[v_i], ..., O_{i,k}[v_i])$, j = 1 to k. It follows from the definition of equivalent executions that all the input values to and the output values from T_i under z and z^* are identical. Therefore, the post-condition of T_i will be either satisfied under both z and z^* or not satisfied under both z and z^*. □

We now prove that setwise serial schedules are consistent and correct. The proof is organized into three Lemmas. Let T_i be a consistent and correct transaction. In Lemma 5-1, we prove that T_i preserves the consistency of each of the accessed atomic data sets and produces correct results when executing alone. This result is valid even if the database as a whole is inconsistent. In Lemma 5-2, we further prove that at the end of executing a transaction ADS segment $\Psi(i, \mathcal{A})$ of T_i, the consistency of $\mathcal{A}$ has been already preserved. In addition, the output values of data objects in $\mathcal{A}$ are correct at the end of $\Psi(i, \mathcal{A})$. We need not wait for the end of T_i to know these results. In Lemma 5-3, we relax the executing alone condition. We show that the results of Lemma 5-2 are still valid for any ADS $\mathcal{A}$, as long as $\mathcal{A}$ is consistent at the beginning of transaction segment $\Psi(i, \mathcal{A})$.

Let $Q = \{\mathcal{A}_1, ..., \mathcal{A}_k\}$ be a given CP partition of D.

Definition: An ADS $\mathcal{A}_j$ is said to be accessed by a transaction, if this transaction reads or writes one or more data objects in $\mathcal{A}_j$.

Lemma 5-1: Let T_i be a consistent and correct transaction. If T_i executes alone and if the states of the atomic data sets accessed by T_i are initially consistent, then at the end of T_i the state of each of the accessed atomic data sets is consistent, and the values output by T_i satisfy the post-condition of T_i.

Proof: let $\mathcal{A}_{i,j}$, j = 1 to k_i, be the atomic data sets accessed by transaction T_i. Let $Y \in \Omega$ be a state of D such that $C_{i,j}(\pi_{S_{i,j}}(Y)) = 1$, j = 1 to k_i; where $C_{i,j}$ represents the ADS consistency constraints of $\mathcal{A}_{i,j}$, and $S_{i,j}$ represents the index set of $\mathcal{A}_{i,j}$. Now let X be a consistent state of the database such that $\pi_{S_{i,j}}(X) = \pi_{S_{i,j}}(Y)$, j = 1 to k_i. Next, we let T_i execute alone with database initially in state X. We now prove that with either X or Y as initial state, the executions of T_i are equivalent.

By assumptions A1-A3, with X as initial state, T_i produces correct results and preserves the consistency of

the database. It follows from theorem 1 that T_i preserves the consistency of all the atomic data sets. Let the values of the local variable and the data object in the execution with initial state X be $L^*_{t_{i,l}}$ and $O^*_{t_{i,l}}$; and that with initial state Y be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We must prove that $L_{t_{i,l}} = L^*_{t_{i,l}}$ and $O_{t_{i,l}} = O^*_{t_{i,l}}$ at each step of the transaction. Recall that the syntax of a transaction step is as follows,

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$$

Since the initial states of all accessed ADS are equal with either X or Y as the initial state, it follows that the initial values of the accessed data objects are equal. Hence, at the first step of T_i, $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because step two either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, h = 1 to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step r+1 either reads the initial value of a data object or a data object which has been output by some step between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. By induction, the final values of accessed data objects with either X or Y as initial state are equal. Since the values of data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, not accessed by T_i remain unchanged, they must be equal at the end of the transaction with either X or Y as the initial state. Let W_x and W_y be the state of D at the end of executing T_i with X and Y as initial states respectively. We have $\pi_{S_{i,j}}(W_x) = \pi_{S_{i,j}}(W_y)$, j = 1 to k_i. The execution using X as the initial state is assumed to preserve the consistency of each accessed atomic data sets; so must be the execution using Y as the initial state. Since the two executions of T_i are equivalent and the execution with X as initial state produces correct results; so must be the execution using Y as initial state. □

There are two implications from this Lemma. First, after the decomposition of the database, a programmer is only required to know and maintain the consistency constraints of the atomic data sets accessed by his transaction. Without a CP partition, everyone must, in principle, know all the database consistency constraints in order to verify that one's transaction will not violate any of them. Second, this Lemma implies that a transaction can still function properly as long as the accessed atomic data sets are consistent, even if the rest of the atomic data sets are inconsistent. This is useful in recovery management, although this topic is outside the scope of this paper.

Lemma 5-2: Let the atomic data sets accessed by transaction T_i be $\mathcal{A}_{i,j}$, j = 1 to k_i. If $\mathcal{A}_{i,j}$, j = 1 to k_i, are initially consistent and if T_i executes alone, then at the end of transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$ the

consistency of $\mathcal{A}_{i,j}$ is preserved. Furthermore, the values of data objects in $\mathcal{A}_{i,j}$ output by $\Psi(i, \mathcal{A}_{i,j})$ are correct at the end of $\Psi(i, \mathcal{A}_{i,j})$.

Proof: At the end of the transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$, $j = 1$ to k_i, the data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i are neither read or written again. It follows that the values of the data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are the same as at the end of the transaction. By Lemma 5-1, at the end of the transaction, the consistency of each of the atomic data sets is preserved, and the values of the data objects output by T_i are correct, so must be at the end of each of the transaction ADS segments. □

Lemma 5-3: In a setwise serial schedule, if at the beginning of a transaction ADS segment, $\Psi(i, \mathcal{A}_{i,j})$, $= 1$ to k_i, ADS $\mathcal{A}_{i,j}$ is initially consistent, then $\mathcal{A}_{i,j}$ is consistent at the end of $\Psi(i, \mathcal{A}_{i,j})$, and the values of each of the data objects in $\mathcal{A}_{i,j}$ output by T_i are correct.

Proof: Let the atomic data sets accessed by T_i be $\mathcal{A}_{i,j}$, $j = 1$ to k_i. Now let T_i execute alone in a serial schedule z^* with the initial states of $\mathcal{A}_{i,j}$, $j = 1$ to k_i, being identical to the initial states of $\mathcal{A}_{i,j}$, $j = 1$ to k_i, in the setwise serial schedule z.

Let the values of the local variables and data objects in the serial schedule z^* be $L^*_{t_{i,l}}$ and $O^*_{t_{i,l}}$; and those in setwise serial schedule z be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We now prove that the executions of T_i under z and z^* are equivalent. Recall that the syntax of a transaction step is given by

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$$

Since the initial states of ADS $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are equal in both schedules, the initial values of all the data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are equal. Therefore, the first steps in both schedules input the same value, i.e. $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because step two either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, $h = 1$ to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step $r+1$ either reads the initial value of a data object or a data object which has been output by some steps between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. Therefore, the final values of accessed data objects in both schedules are equal at the end of each transaction ADS segment. In addition, data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, not accessed by $T_{i,j}$ remain unchanged and therefore equal at the end of each transaction ADS segment for both schedules. It follows from lemma 5-2 that at the end of $\Psi(i, \mathcal{A}_{i,j})$, the $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are consistent, and the value of each of the data object in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, output by T_i is correct. □

Theorem 5: A setwise serial schedule is consistent and correct.

Proof: Let $Q = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ be a CP partition of D. Let the initial states of each of the ADS's be $Z_{A_j}[0]$, j = 1 to k. These initial states are assumed to be consistent.

Since a schedule is a totally ordered set of steps from all the transactions, each of which terminates, there must exist a transaction ADS segment $\Psi(i, \mathcal{A}_j)$ which first finishes its computation. Let the associated ADS state be $Z_{A_j}[1]$. Since there is no interleavings among transaction ADS segments accessing the same ADS in a setwise serial schedule, $Z_{A_j}[1]$ must be output by a transaction which has used only the initial states that were assumed to be consistent. By Lemma 5-3, $Z_{A_j}[1]$ is consistent, and the values of data objects in $\mathcal{A}_j$ output by $\Psi(i, \mathcal{A}_j)$ are correct. Consider now the output of the second transaction ADS segment produced by the schedule. Since it can use only $Z_{A_j}[1]$ or $Z_{A_m}[0]$, m = 1 to k and $m \neq j$, at the end of this second transaction ADS segment, the accessed atomic data set is in a consistent state and the output values are correct by Lemma 5-3. Now assume that the first n transaction ADS segments produce consistent and correct results. The $n+1^{st}$ must also by the same argument. By induction, the ADS state produced by each of the transaction ADS segments is consistent, and the values of the data objects output in each ADS at the end of the transaction ADS segment satisfy the post-condition. It follows that a setwise serial schedule is consistent and correct. □

Corollary 5: Setwise serializable schedules are consistent and correct.

Throughout this section, the choice of atomic data sets has been arbitrary. This is because the theorems apply to any CP partition whether maximal or not. If the CP partition consists of a single ADS, then setwise serializable schedules reduce to serializable schedules.

3.3.3.1.4 Algorithms for Maintaining Setwise Serializability

We have shown that if the database has been partitioned into consistency preserving atomic data sets, then a setwise serializable schedule is consistent and correct. To enforce setwise serializability, we only need to slightly modify the algorithms developed for the serializability theory. For example, we can modify the two phase lock protocol [Eswaren 76] as follows.

Definition: A setwise two phase lock protocol requires a transaction not to release any lock on any data object of an atomic data set until all the locks in this atomic data set have been acquired. Once any lock in an atomic data set has been released, no more data objects in the same atomic data set can be locked.

Theorem 6: A setwise two phase lock protocol guarantees setwise serializability.

Proof: Suppose that the claim is false. Then at least one of the ADS precedence graph contains a cycle, such

as $T_1 > T_2 > T_k > T_1$. This implies that a lock of T_1 follows an unlock of T_1. This contradicts the assumption that the locking protocol is setwise two phase. □

Finally, we want to comment on the possible structures of an atomic data set. It is not necessary that the structures be a single level. An atomic data set, like a general purpose database, can have structure. For example, an atomic data set can have a tree structure. In this case, the tree protocol [Silberschatz 80] can be used to enforce setwise serializability. This protocol requires that,

- except for the first item locked, no item can be locked unless a lock is currently held on its parent.
- no item is ever locked twice.

Note that the first item need not be the root, and the locking need not to be two phase.

3.3.3.2 Nested Transactions

In the early work on serializability theory, a transaction was modelled as a sequence of steps. However, it is natural to write transactions in a nested form, in which sub-transactions can be executed in parallel and invoked by higher level ones. Recently, serializable nested transactions have been studied by [Gray 81, Moss 81, Lynch 83b, Beeri 83]. From a concurrency control point of view, the new issue associated with serializable nested transactions is how to provide an "executing alone" environment for a parallel program. This can be illustrated by the "lock passing" problem among parallel sub-transactions. Suppose that a data object is shared by several sub-transactions of the same transaction. A sub-transaction which first accesses this data object must be able to pass the "lock" to other sub-transactions. If it releases the "lock" to other transactions, the rest of sub-transactions needing this object may face unpredictable modifications to this data object caused by other transactions. The nested transaction, as a whole, can no longer be considered to be executing alone.

Given a nested transaction, the setwise serializable scheduling rule partitions the steps of the transaction into transaction ADS segments. Due to the nested structure, a transaction ADS segment could be distributed in several sub-transactions that can be executed parallelly. We prove that the consistency of the database will still be preserved and the post-condition of each of the transactions will be still be satisfied as long as the schedule for the transaction system is setwise serializable.

3.3.3.2.1 Syntax

We can visualize a nested transaction as being organized in the form of a tree. Nodes in the tree are sub-transactions and leaves are steps. The execution of the transaction is defined by the partial order of the tree.

Definition: A nested transaction, T_i^N, is a partially ordered set of steps with the following syntax:

<NestedTransaction> ::= BeginTransaction <NestedTransactionBody> EndTransaction.

<NestedTransactionBody> ::= BeginSerial <SubTransactionList> EndSerial |
BeginParallel <SubTransactionList> EndParallel

<SubTransactionList> ::= <SubTransaction> | <SubTransactionList><SubTransactionList>

<SubTransaction> ::= <Step> | <SubTransaction>;<SubTransaction>; | <NestedTransactionBody>

<Step> ::= ReadStep | WriteStep

A step, either "read" or "write", is modelled as an indivisible execution of the following two instructions.

$$L_{t_{i,j,k}} := O_{t_{i,j,k}}$$

$$O_{t_{i,j,k}} := f_{t_{i,j,k}}(\{L_{t_{i,j,k}}\} \cup \{L_{t_{i,m,n}} \mid t_{i,m,n} < t_{i,j,k}\})$$

where $t_{i,j,k}$ is the k^{th} step at level j of transaction T_i^N, $L_{t_{i,j,k}}$ is the local variable used by step $t_{i,j,k}$, $O_{t_{i,j,k}}$ is the data object accessed by step $t_{i,j,k}$, and $f_{t_{i,j,k}}$ represents the computation performed by step $t_{i,j,k}$. Note that step $t_{i,j,k}$ can use its own local variable and local variables associated with steps preceeding it in the partial ordering of transaction steps. A "read" step is interpreted as one which writes the original value back into the data object i.e. $f_{t_{i,j,k}}$ is the identity function.

3.3.3.2.2 Consistency and Correctness

Due to the partial ordering, the ordering between some steps in the transaction is unspecified. The results produced by *any total ordering that is consistent with the partial ordering* in the transaction must be equally valid. Otherwise, one should specify the order.

Definition: A single level transaction T_i^S is the linearization of the nested transaction T_i^N, if and only if T_i^S has the same steps as T_i^N and if the the total ordering of steps in T_i^S is consistent with the partial ordering of steps in T_i^N. That is,

$$[\forall(t)((t \in T_i^S) \Leftrightarrow (t \in T_i^N))] \wedge [\forall((t_k, t_m \in T_i^N) \wedge (t_m > t_k))((t_k, t_m \in T_i^S) \wedge (t_m > t_k))]$$

Definition: A nested transaction is said to be consistent and correct, if and only if each of the linearizations of the nested transaction, when executed alone satisfies our three assumptions about a transaction: it terminates (A1), preserves the consistency of the database (A2), and produces correct results(A3). In the following, we limit our investigation only to consistent and correct nested transactions.

We now define the notion of a nested transaction system and its schedules.

Definition: A nested transaction system $T^N = \{T_1^N, ..., T_m^N\}$ is a finite set of nested transactions operating upon the shared database D.

Definition: Let T_i^S be the set of all the linearizations of nested transaction $T_i^N \in T^N$. Let T^S be a linearized transaction system for T^N. That is, $T^S = \{T_1^S, ..., T_m^S\}$ is a transaction system in which $T_i^S \in T_i^S$, i = 1 to m. Let T^S be the set of all the linearized transaction systems for T^N. A schedule z for a nested transaction system T^N is a a schedule of a linearized transaction system $T^S \in T^S$.

Theorem 7: Setwise serial schedules of a nested transaction system are consistent and correct.

Proof: By definition, each of the linearizations of a nested transaction, when executing alone and when the database being initially consistent, terminates, preserves the consistency of the database and produces correct results. It follows from Theorem 5 that a setwise serial schedule for a linearized nested transaction system is consistent and correct. This is true for all the linearizations of the given nested transaction system. It follows that setwise serializable schedules for nested transaction systems are consistent and correct. □

Corollary 7: Setwise serializable schedules for nested transactions are consistent and correct.

To implement a setwise two phase lock for a nested transaction, the principle is to ensure that the setwise two phase lock protocol is observed among transactions while permitting internal lock passing within a nested transaction. This can be done by following Moss' lock passing method [Moss 81]. Each sub-transaction follows the setwise two phase lock protocol. However, locks released by sub-transactions are retained by their parent. These locks can be acquired by other sub-transactions under that parent, but not by other transactions. After the parent releases any lock on an atomic data set, none of its children can acquire any new lock on this atomic data set. A given level L in a nested transaction is said to be the top level for ADS $\mathcal{A}$, if level L does not pass locks on $\mathcal{A}$ to higher levels and if the locks on $\mathcal{A}$ directly acquired at level L plus those retained from lower levels constitute the complete set of locks on ADS $\mathcal{A}$. Data objects in an atomic data set can be unlocked only at the top level with respect to this atomic data set.

3.3.3.3 Compound Transactions

The setwise serializable scheduling rule does not use any semantic information to guide the partition of individual transactions. It takes a transaction and partition its steps into transaction ADS segments, independent of the semantics of the transaction. To obtain a higher degree of concurrency, the semantic information of one's own transaction must be utilized in the scheduling process. Generalized setwise serializable scheduling rules are a family of modular scheduling rules designed for this purpose. These rules are

represented by the new transaction syntax called *compound transactions*. In other words, users of this family of rules must carefully study their own transactions and try to express their transactions in the form of compound transactions.

In a compound transaction, steps are partitioned into equivalent classes called elementary transactions, each of which terminates, preserves the consistency of the database and produces results satisfying its own post-condition. In a compound transaction, elementary transactions are partially ordered, and the conjunction of the post-conditions of the elementary transactions must be equivalent to the post-condition of the compound transaction. Once a transaction is expressed in the form of compound transactions, each of the elementary transactions in a compound transaction can be further partitioned into transaction ADS segments. A schedule z is said to be generalized setwise serializable if under z all the transaction ADS segments of all the elementary transactions in the system are executed serializably. In this section, we define the notion of compound transactions and prove that generalized setwise serializable schedules are consistent and correct.

Before the development of a formal model, we would like to illustrate the concepts with a simplified examples of resource management.

3.3.3.3.1 Consistency Preserving Partition of Transactions --- An Example

Suppose that a distributed computer system consists of nodes having various resources. These resources are described by counter variables which indicate the units of various resources available and lists which describe the units loaned to various processes. For simplicity, we only consider a single type of resource at each node. The counter variable and the list at each node form an atomic data set with consistency constraint requiring the sum of the units of the available resources and the loaned resources to be a constant. Let the counter variable and the list at node i be K_i and L_i respectively. Consider a transaction, T_j, which attempts to obtain one unit of resources at both nodes 1 *and* 2, or none at all. Without using the idea of compound transaction, we code T_j in the form of a nested transaction. To illustrate the locking protocol, we write the following pseudo-code in which sub-transactions are written redundantly and in line.

```
Nested Transaction Tj
Data Objects: K1, K2, L1, L2;
BeginTransaction
BeginSerial
  BeginParallel
    WriteLock K1;
    WriteLock K2;
  EndParallel;
  if not ((K1 > 0) and (K2 > 0)) then
  BeginParallel
    Unlock K1;
    Unlock K2;
  EndParallel
                                  else
```

```
  BeginParallel
    BeginSerial
      Sub-Transaction GetResource1
      BeginSerial
        K1 := K1 - 1;
        WriteLock L1;
        Update L1;
      EndSerial; {end of sub-transaction}
      Unlock L1;
      Unlock K1;
    EndSerial;

    BeginSerial
      Sub-Transaction GetResources2
      BeginSerial
        K2 := K2 - 1;
        WriteLock L2;
        Update L2;
      Endserial; {end of sub-transaction}
      Unlock K2;
      Unlock L2;
    EndSerial;
  EndParallel;
EndSerial;
EndTransaction.
```

This provides a higher degree of concurrency than that permitted by a serializable schedule because locks on each atomic data set are released as soon as the operations on each set are done, even if the transaction has not obtained all the locks. However, such an approach may not provide enough concurrency when the communication delay among nodes is large and when the transaction tries to get resources from many different nodes. This is because the transaction must obtain all the locks on K_i, i = 1 to n, before it can decide if it can proceed. This could block the system resource allocation activity for a significant amount of time.

Fortunately, the degree of concurrency can be markedly increased by rewriting T_j as a compound transaction. For the purpose of illustrating the locking protocol, we write the following pseudo-code in which elementary transactions are written redundantly and in line.

```
Compound Transaction Tj
Data Objects: K1, K2, L1, L2;
Local  Variables: ObtainResource1, ObtainResource2;
BeginTransaction
BeginSerial
  BeginParallel
    Elementary Transaction GetResource1
    BeginSerial
      ObtainResource1 := false;
      WriteLock K1;
      if K1 > 0 then
      BeginSerial
```

```
        K1 := K1 - 1;
        ObtainResource1 := true;
        WriteLock L1;
        Update L1;
        Unlock L1;
      EndSerial;
      Unlock K1;
    EndSerial; {end of elementary transaction}

    Elementary Transaction GetResource2
    BeginSerial
      ObtainResource2 := false;
      WriteLock K2;
      If K2 > 0 then
      BeginSerial
        K2 := K2 - 1;
        ObtainResource2 := true;
        WriteLock L2;
        Update L2;
        Unlock L2;
      EndSerial;
      Unlock K2;
    EndSerial; {end of elementary transaction}
  EndParallel;

  BeginParallel
    Elementary Transaction ReturnResource1
    BeginSerial
      If (ObtainResource1) and not (ObtainResource2) then
      BeginSerial
        WriteLock K1;
        K1 := K1 + 1;
        WriteLock L1;
        Update L1;
        Unlock K1;
        Unlock L1;
      EndSerial;
    EndSerial; {end of elementary transaction}

    Elementary Transaction ReturnResource2
    BeginSerial
      if (ObtainResource2) and not (ObtainResource1) then
      BeginSerial
        WriteLock K2;
        K2 := K2 + 1;
        WriteLock L2;
        Update L2;
        Unlock K2;
        Unlock L2;
      EndSerial;
    EndSerial; {end of elementary transaction}
  EndParallel;
EndSerial;
EndTransaction.
```

Note that in this example, each elementary transaction follows the setwise two phase lock protocol but the the compound transaction does not. Since the compound transaction violates the setwise two phase lock protocol, we cannot use Corollary 5 to conclude that it will maintain the consistency of the database and produce correct results. Possible inconsistency or incorrectness seemingly could arise, because certain data objects used by a compound transaction could be modified by other transactions during the execution of the compound transaction. For example, the unlocking and relocking of K_1 and L_1 permits other transactions to assign any arbitrary but consistent values to K_1 and L_1 during the execution of the compound transaction T. Nevertheless, the consistency and correctness of a compound transaction follows from our consistency and correctness assumptions about elementary transactions as well as our new assumption that a compound transaction produces correct results if each of its elementary transactions produces correct results. We formalize these ideas as follows.

3.3.3.3.2 Syntax

We can visualize a compound transaction as being a tree with the nodes being sub-compound transactions and the leaves being elementary transactions. Each elementary transaction has the structure of a nested transaction.

Definition: A compound transaction is a partially ordered set of elementary transactions defined as follows.

<CompoundTransaction> ::= BeginTransaction <CompoundTransactionBody> EndTransaction.

<CompoundTransactionBody> ::= BeginSerial <SubCompoundTransactionList> EndSerial |
BeginParallel <SubCompoundTransactionList> EndParallel

<SubCompoundTransactionList> ::= <ElementaryTransaction> | <CompoundTransactionBody> |
<SubCompoundTransactionList>;<SubCompoundTransactionList>

<ElementaryTransaction> ::= NestedTransactionBody[1].

3.3.3.3.3 Consistency and Correctness

Having defined the syntax of a compound transaction, we must consider a system of compound transactions and determine the set of schedules which are consistent and correct.

Assumption: Each elementary transaction, terminates (A1), preserves the consistency of the database (A2) and satisfies its post-condition (A3) when executing alone and when the database is initially consistent.

---

[1]Defined in Section 3.2.3.1.a

Definition: The post-condition of a compound transaction is equivalent to the conjunction of the post-conditions of its elementary transactions.

Definition: A compound transaction system $T^c = \{ T_1^c, T_2^c, ..., T_n^c \}$ is a finite set of compound transactions operating on database D.

Definition: A schedule z of a compound transaction system T^c is a totally ordered set of all the steps in T^c, such that the ordering of steps of each compound transaction T_i^c, i = 1 to n, in the schedule is consistent with the partial ordering of these steps in transaction T_i^c, i = 1 to n.

Let $t_{i,j} > t_{k,m}$ denote that step $t_{i,j}$ is executed after $t_{k,m}$.

$$[\forall (t)((t \in z) \Leftrightarrow (t \in T^c))] \wedge [\forall (T_i^c \in T^c) \forall ((t_{i,j}, t_{i,k} \in T_i^c) \wedge (t_{i,k} > t_{i,j}))((t_{i,j}, t_{i,k} \in z) \wedge (t_{i,k} > t_{i,j}))]$$

Definition: An elementary transaction system T^e is said to be associated with the compound transaction system T^c if and only if,

$$\forall (T_i^e)((T_i^e \in T^e) \Leftrightarrow (T_i^e \in T^c))$$

where T_i^e is an elementary transaction of T^c.

Definition: A schedule of a compound transaction system is said to be *generalized setwise serializable* if and only the associated elementary transaction system is setwise serializable.

Theorem 8: Generalized setwise serializable schedules are consistent and correct.

Proof: Since the schedule is setwise serializable with respect to all the elementary transactions in the system, it follows from Corollary 5 and the definition of elementary transactions that each elementary transaction terminates, preserves the consistency of the database and produces results that satisfy its post-condition. Hence, the consistency of the database is preserved. By definition, the post-condition of each of the compound transactions is also satisfied. Hence, generalized setwise serializable schedules are consistent and correct. □

Finally, it follows from the definition that generalized setwise serializable schedules can be implemented by requiring each of the elementary transactions in the transaction system to follow the setwise two phase lock protocol.

3.3.4 Modularity, Application Independence and Optimality

In this section, we first formalize the important concepts of "modularity" and "application independence". Having set up the theoretical framework, we prove that setwise serializable schedules are optimal in the set of application independent schedules and that generalized setwise serializable schedules form a complete class in the set of modular schedules.

3.3.4.1 Modularity and Application Independence

A transaction facility consists of a set of transactions operating upon a shared database. For the remainder of this section, we assume that in the design phase the consistency constraints of the database are specified, and the resulting consistency preserving partition is determined and remains *fixed*. Programmers are then required to write transactions for various applications that use the system database and observe the database consistency constraints. Having written or modified his transaction, the programmer must schedule his transaction according to some rule so that transactions can be executed concurrently, consistently and correctly.

A transaction scheduling rule is a specification of the permissible interleaving of the steps of a given transaction with the steps of other transactions. Given a transaction system, a transaction scheduling rule partitions the steps of each transaction into *atomic step segments* that will be executed without being interferred with by steps of other transactions. For example, in serializable schedules, all the steps in a single transaction are grouped into a single atomic step segment. In setwise serializable schedules, each transaction ADS segment (steps accessing the same ADS) is taken as an atomic step segment. In generalized setwise serializable schedules, the transaction ADS segments in each of the elementary transactions are atomic step segments. Once the partition of the steps in a transaction has been specified, one can use "locks", "time-stamps" or other protocols to ensure that steps from various transactions are interleaved in such a way that each atomic step segment will be executed serializably.

In the transaction system semantic information approach, one is allowed to utilize all the information about the given *transaction system* to schedule each transaction in the system. For example, let the database D = {A, B} with consistency constraints "A + B = 100". Suppose that $\{T_1, T_2\}$ is a transaction system where T_1 = {A := A -1; B := B + 1} and T_2 = {B := B - 2; A := A + 2}. After examining the details of these two transactions, one may determine that the appropriate atomic partition of T_i, i = 1 to 2, is to specify each step as an atomic step segment. That is, steps of T_1 and T_2 can be interleaved arbitrarily. On the other hand, in another related transaction system $\{T_1, T_2^*\}$ where T_2^* = {B := B - 2; A := 100 - B}, the correct specification requires the entire transaction T_1 (T_2^*) to be treated as a single atomic step segment, even though T_2 and T_2^* are equivalent when executing alone. That is, T_1 and T_2^* must be interleaved serializably. In a large transaction system, a transaction system semantic information approach often requires users to first partition the trans-

action system into different sub-transaction systems. Each transaction is then partitioned into different forms each of which is suitable for a given sub-transaction system. For example, a nested form of multiple partitions specified by "break points" was suggested in [Lynch 83a]. In contrast to the transaction system semantic information approach, a modular approach requires that the atomic partition of each transaction be constructed independent of the transaction system, so that the modification of any transaction will not invalidate the atomic partition of another transaction.

Before proceeding, we first define the notion of a scheduling rule. A scheduling rule is a function which takes a transaction system and partitions the steps of each transaction into equivalent classes called *atomic step segments.*

Definition: Let $\mathcal{T}_m$ denote the set of all the possible consistent and correct transactions with m steps. Let $\mathcal{T}$ denote the set of all the possible consistent and correct transactions, i.e. $\mathcal{T} = \cup_{m=1}^{\infty} \mathcal{T}_m$. Let P_m denote a partition of an m-step consistent and correct transaction into atomic steps segments. Let $\mathcal{P}_m$ denote the set of all the possible partitions of an m-step consistent and correct transaction. Let $\mathcal{P}$ be the set of all the possible partitions, i.e. $\mathcal{P} = \cup_{m=1}^{\infty} \mathcal{P}_m$.

A scheduling rule for a transaction system with n transactions, R_n, is a function which takes the transaction system of size n and partitions each of the n transactions,

$$R_n: \prod_{i=1}^{n} \mathcal{T} \rightarrow \prod_{i=1}^{n} \mathcal{P}$$

A scheduling rule R is a function with takes a transaction system of any size and partitions each of the transactions in the system.

$$R: \cup_{n=1}^{\infty} (\prod_{i=1}^{n} \mathcal{T}) \rightarrow \cup_{n=1}^{\infty} (\prod_{i=1}^{n} \mathcal{P})$$

such that the restriction of R to $\prod_{i=1}^{n} \mathcal{T}$ is R_n, i.e.

$$R|\prod_{i=1}^{n} \mathcal{T} = R_n, \; n = 1 \text{ to } \infty$$

Given a scheduling rule R, we must identify the set of schedules that satisfy R. A schedule z satisfies R if each of the atomic step segments specified by R is executed serializably under z. This is formalized as follows.

Definition: Let $T = \{T_1, ..., T_n\}$ be a consistent and correct transaction system, i.e. $T \subset \mathcal{T}$. Let T_i be a transaction in T. Let $\Xi_R(T_i)$ denote the atomic partition of the steps of T_i by R_n, the restriction of R to $\prod_{i=1}^{n} \mathcal{T}$. Let Γ be the set of all the atomic step segments of T specified by R, i.e. $\Gamma = \cup_{T_i \in T} \Xi_R(T_i)$. Let D be the database and $Z(T)$ be the set of all the possible schedules for T.

A schedule $z \in Z(T)$ is said to satisfy R if and only if,

$$Cycle(G(z, \Gamma, D)) = 0,$$

where $G(z, \Gamma, D)$ is the precedence graph[2] for schedule z with respect to the database D and the set of step segments Γ. The set of all such schedules for T is denoted by $Z_R(T)$. That is, $Z_R(T) = \{z \mid Cycle(G(z, \Gamma, D)) = 0\}$

Definition: A scheduling rule R is said to be consistent and correct, if and only if all the schedules that satisfy R are consistent and correct,

$$\forall (T \subset \mathbb{T}) \forall (z \in Z_R(T))\ (z \text{ is consistent and correct})$$

In the following, we limit our discussions to consistent and correct scheduling rules. We consider a consistent and correct scheduling rule to be *modular*, if it schedules each transaction independent of other transactions in the system.

Definition: A consistent and correct scheduling rule R is said to be *modular* if and only if R schedules each transaction independently, i.e.

$$R_n(\{T_1, ..., T_n\}) = (R_1(\{T_1\}), ..., R_1(\{T_n\})),\ n = 1 \text{ to } \infty,$$

where R_n is the restriction of R to $\prod_{i=1}^{n} \mathbb{T}$. The scheduling rule for individual transactions, R_1, will be referred to as the kernel of the modular scheduling rule R.

We now turn to the concept of an application independent scheduling rule. Thus far we have assumed that each programmer writes and schedules his own transaction. The scheduling is done with full knowledge of the transaction written but without specific knowledge of others' transactions. To further simplify the scheduling task, we would like to develop scheduling rules that can be mechanically applied to all the transactions, independent of their semantics. To this end, an application independent scheduling rule must ignore the specifics of various transactions and use only the syntax information of transactions, i.e. names of the data objects read or written by each transaction. In other words, an application independent scheduling rule views a transaction as a sequence of read and write steps without knowing the computation carried out by the transaction.

---

[2]Defined in Section 3.2.3.1.b

Definition: Define an equivalence relation on the set of all the consistent and correct transactions T. Two consistent and correct transactions T_i and T_j are said to be *equivalent in syntax*, denoted by $T_i \equiv T_j$, if and only if,

1. T_i and T_j have the same number of steps.
2. If step k of T_i reads (writes) data object O, then step k of T_j reads (writes) the same data object O, for all k.

Definition: A modular scheduling rule R is said to be *application independent* if the kernel of R identically partitions transactions with equivalent syntaxes.

$$\forall ((T_i, T_j \in T) \wedge (T_i \equiv T_j))\,(R_1(T_i) = R_1(T_j))$$

Theorem 9: Setwise and generalized setwise serializable schedules are modular.

Proof: First, setwise serializable schedules are a special case of generalized setwise serializable schedules. Second, generalized setwise serializable schedules are consistent and correct by Theorem 8. Third, a generalized setwise serializable schedule takes each transaction separately and partitions it into elementary transactions and then partitions the elementary transactions into ADS transaction segments. This is done independently for each transaction. It follows from the definition of modular scheduling rules that both scheduling rules in question are modular. □

Theorem 10: The setwise serializable scheduling rule is application independent.

Proof: A setwise serializable scheduling rule partitions the steps of the given transaction into transaction ADS segments. A transaction ADS segment consists of all the steps which read or write data objects from the same ADS. It follows that two transaction which are equivalent in syntax have the same partition. It follows from the definition that a setwise serializable scheduling rule is application independent. □

It should be pointed out that a generalized setwise serializable scheduling rule is modular but not application independent. This is because the decomposition of a transaction into a collection of elementary transactions requires an understanding of the details of the transaction in question. We cannot correctly perform the decomposition using syntax information alone.

3.3.4.2 Optimality and Completeness

The set of primitive steps used in a given syntax affects the degree of concurrency provided by a scheduling rule. The primitive steps defined in our transaction syntax are the conventional two: "read" and "write". It has been shown that for serializable schedules the concurrency can be improved if the set of primitive steps is expanded to include other commutative ones [Korth 83]. The idea is that if a set of steps is commutative, then there is no need to control their relative order. For example, read steps are commutative with each other, as are unconditional add steps. For a full treatment of this subject, readers are referred to [Korth 83]. In the following, we limit our discussion to transactions using only primitive steps: "read" and "write". The use of commutative steps to improve the concurrency of (generalized) setwise serializable schedules can be done in a manner similar to that done by Korth for serializable schedules.

We begin our investigation by first defining a way to compare the degree of concurrency offered by different scheduling rules.

Definition: Scheduling rule R^1 is said to be *at least as concurrent as* R^2, denoted by $R^1 \geq R^2$, if and only if,

$$\forall (T \subset \mathcal{T})(Z_{R^2}(T) \subseteq Z_{R^1}(T))$$

That is, the concurrency of schedules is partially ordered by set containment. The relative concurrency of two scheduling rules can be incomparable. We now define the notion of optimal application independent scheduling rules.

Definition: Let Λ_A be the set of all the application independent scheduling rules. An application independent scheduling rule $R^* \in \Lambda_A$ is said to be optimal if R^* is at least as concurrent as any rule in Λ_A. That is,

$$\forall (R \in \Lambda_A)(R^* \geq R)$$

We now prove that the setwise serializable scheduling rule is optimal in the set of application independent scheduling rules. The key to the proof is to show that if a modular scheduling rule R partitions a transaction ADS segment σ into two or more atomic step segments, then there exists some schedule z satisfying R such that the inputs to σ under z cannot come from a single consistent state.

Lemma 11-1: Let $\mathcal{A} = \{O_1, ..., O_m\}$ be an ADS from the maximal consistency preserving partition of D. Let the index set of $\mathcal{A}$, I, be partitioned into two sets S_1 and S_2. Let U be the universal set of the consistent states of $\mathcal{A}$.

$$\exists (W \in U \wedge X \in U \wedge Y \notin U)((\pi_{s_1}(W) = \pi_{s_1}(Y)) \wedge (\pi_{s_2}(X) = \pi_{s_2}(Y)))$$

Proof: Suppose that the claim is false. By the definition of a consistency preserving partition, $\{S_1, S_2\}$ is a consistency preserving partition of I. This contradicts the assumption that D is maximally partitioned. □

Lemma 11-2: Let database D be maximally partitioned into atomic data sets. Let T_i be a consistent and correct transaction operating upon D. Let σ be a transaction ADS segment in T_i. Let the ADS accessed by σ be $\mathcal{A}$. Suppose that σ is partitioned into atomic step segments $\sigma_1, ..., \sigma_k$, $k > 1$ by some modular scheduling rule R. Then there exist a transaction system $T \in \mathbf{T}$ and a schedule $z \in Z_R(T)$ such that under z the values input to σ are not projections from any single consistent state of $\mathcal{A}$.

Proof: Let $\Xi_{ADS}(T_i) = \{\sigma_1, ..., \sigma_k\}$ be the ADS atomic partition of T_i. That is, each element of the partition is a transaction ADS segment. Suppose that a modular scheduling rule R partitions transaction ADS segment σ_1 into $\sigma_{1,1}, ..., \sigma_{1,j}$, $j \geq 2$. In the following, we prove that if $j = 2$ then the inputs to σ cannot come from a single consistent state of $\mathcal{A}$. If $j > 2$, we merge $\sigma_{1,2}, ..., \sigma_{1,j}$ into a single atomic step segment σ_2^* and then use the result for $j = 2$. There can be two cases.

Case 1: $\sigma_{1,1}$ and $\sigma_{1,2}$ access disjoint sets of data objects in ADS $\mathcal{A}$. Let I be the index set of $\mathcal{A}$. Partition I into S_1 and S_2 such that σ_i accesses only data objects indexed by S_i, $i = 1$ to 2.

Let T_w and T_x be two transactions $\in \mathbf{T}$, assigning consistent states W and X to $\mathcal{A}$ respectively. Let $W_1 = \pi_{S_1}(W)$ and $X_2 = \pi_{S_2}(X)$. By Lemma 11-1, W_1 and X_2 could be the projection of an inconsistent state Y. We assume this is the case. The schedule $z = \{T_x, \sigma_{1,1}, T_y, \sigma_{1,2}, ..., \sigma_k\}$ satisfies R because each of the atomic step segments specified by R is executed serializably under z. Note that the inputs to σ are projections from the inconsistent state Y.

Case 2: $\sigma_{1,1}$ and $\sigma_{1,2}$ share at least one data object. Let a shared data object be O. As discussed in case 1, the schedule $z = \{T_x, \sigma_{1,1}, T_y, \sigma_{1,2}, ..., \sigma_k\}$ satisfies R. Let the values of O assigned by T_x and T_y be x and y respectively. Hence, under z there are two steps accessing O, inputting values from two different states of $\mathcal{A}$. It follows that R cannot guarantee all the inputs to σ_1 are projections from a single consistent state of $\mathcal{A}$. □

Theorem 11: The setwise serializable scheduling rule is optimal in the set of application independent scheduling rules.

Proof: Let the setwise serializable scheduling rule be R^*. Let R be any other application independent scheduling rule. Let T_i be any consistent and correct transaction. Let the database be maximally partitioned into atomic data sets. Let σ be a transaction ADS segment of T_i accessing ADS $\mathcal{A}$. We examine the atomic partition of T_i by R. If any ADS segment σ of T_i is partitioned, then by Lemma 11-1 the inputs to σ cannot

be guaranteed to be the projections of a single consistent state of $\mathcal{A}$. Since R identically partitions all the transactions equivalent to T_i in syntax, we can interpret the semantics of T_i as follows. T_i produces correct results if and only if the input values to σ are consistent, i.e. projections from a single consistent state of $\mathcal{A}$. Hence, if any transaction ADS segment is partitioned by R, R will be incorrect. It follows that the atomic step segments of T_i specified by R can only be either transaction ADS segments or the super sets of transaction ADS segments. This result applies to any transaction T_i in any transaction system $T \subset \mathbb{T}$. It follows that any schedule z satisfying R satisfy R^*. Therefore, R^* is at least as concurrent as R. □

We now investigate the completeness issue of generalized setwise serializable scheduling rules. Associated with each transaction, there is the specification of input steps, output steps and the relationship among input values and output values in the form of post-conditions. The transaction must be written in such a way that the consistency of the database is preserved and the post-condition is satisfied when executing alone. However, in many cases the isolated execution environment is only a sufficient condition. We have shown that if we are able to partition the steps of a transaction into elementary transactions and then partition the steps of each elementary transaction into transaction ADS segments, the consistency of the database is preserved and the specified post-condition is satisfied.

We have developed the syntactic structure of compound transactions to support users to form such modular but application dependent atomic partition of transactions. The question remaining to be answered is whether there exists another form of partitioning a given transaction that would be modular and lead to a higher degree of concurrency. To answer this question, we introduce the notion of *completeness*. We say that the generalized setwise serializable scheduling rules form a *complete class* within the set of modular scheduling rules. This means that given any modular scheduling rule R we can always find a generalized setwise serializable scheduling rule R^* such that R^* is at least as concurrent as R. Hence, a programmer who is interested in modular scheduling rules providing a high degree of system concurrency needs to look no further than the class of generalized setwise serializable scheduling rules. All he has to do is to maximally partition his transaction into elementary transactions. Once this is done, each of the elementary transactions can be mechanically partitioned into transaction ADS segments.

Definition: Let Λ_M be the set of all the modular scheduling rules. A set of scheduling rules $\mathfrak{R}$ is said to form a *complete class* within Λ_M, if and only if

$$\forall (R \in \Lambda_M)[\exists (R^* \in \mathfrak{R})(R^* \geq R)]$$

Before proceeding with the proof of completeness, we need to introduce the notion of the post-condition associated with an atomic step segment. For example, let the transaction $T_i = \{t_{i1}: A := A - 1; t_{i2}: A := A$

+ 1}. If the two steps of T_i are treated as a single atomic step segment, then $t_{i,1}$ is an input step and $t_{i,2}$ is an output step. The partition of a transaction could create input and output steps in addition to those defined in an executing alone environment. For example, if each of these two steps is an atomic step segment, then $t_{i,1}$ ($t_{i,2}$) is both an input step and an output step.

Definition: Let $\sigma = \{t_{i,1}, ..., t_{i,k}\}$ be an atomic step segment. Let the data object accessed by step $t \in \sigma$ be O. Step $t_{i,j}$ is an input step if it is the step in σ first accessing O. Step $t_{i,j}$ is an output step if it is the step in σ last accessing O.

Definition: Let $O_j[v_i]$, j = 1 to k, be the values input to the input steps of σ and $O_j[v_f]$, j = 1 to k, be the values output by the output steps of σ. The post-condition of σ is a specification of the output values as functions of input values when σ is executing alone.

$O_j[v_f] = f_j(O_1[v_i], ..., O_k[v_i]), j = 1 \text{ to } k.$

We now prove that generalized setwise serializable scheduling rules form a complete class.

Lemma 12: A modular scheduling rule R is consistent and correct if and only if for each of the transactions T_i in T,

1. Each atomic step segment in T_i specified by R preserves the consistency of the database when executing alone.

2. The conjunction of the post-conditions of all the atomic step segments in T_i specified by R is equivalent to the post-condition associated with T_i.

Proof: First, if any atomic step segment σ in T_i specified by R does not preserve the consistency of the database when executing alone, then another transaction T_j executing after σ would input an inconsistent state. Since R is modular, we can define the semantics of T_j as one that outputs incorrect results when its input is inconsistent. Thus R is incorrect. Second, if the conjunction of the post-conditions of all the atomic step segments in T_i specified by R is not equivalent to the post-condition associated with T_i, then R is incorrect by definition. Since any schedule z for any transaction system $T \subset \mathbf{T}$ satisfying R guarantees that each of the atomic step segments will be executed serializably, it follows from condition 1 and 2 that z is consistent and correct. □

Theorem 12: Generalized setwise serializable scheduling rules form a *complete class* within the set of modular scheduling rules.

Proof: Let R be any modular scheduling rule. Suppose that T_i is a consistent and correct transaction and T_i

is partitioned into atomic step segments σ_1, ..., σ_k by R. First, by Lemma 12 σ_i, i = 1 to k, must preserve the consistency of the database when executing alone. Second, by Lemma 12 the conjunction of the post-conditions of σ_1, ..., σ_k must be equivalent to the post-conditions associated with T_i. Note that each σ_i, i = 1 to k, satisfies the definition of an elementary transaction. Define a generalized setwise serializable scheduling rule R^* which partitions T_i as follows. First, R^* labels σ_1, ..., σ_k as elementary transactions. Next, R^* partitions these elementary transactions into transaction ADS segments. Hence, R^* is at least as concurrent as R. □

3.3.5 Conclusion

The very nature of a distributed system provides us with the opportunity to realize a very high degree of concurrency. The desire to realize a higher degree of concurrency than that permitted by serializable schedules has motivated computer scientists to develop non-serializable concurrency control methods. However, a distributed computer system is typically very complex, written and maintained by many programmers over a period of years. Therefore, it is important to develop a modular approach to non-serializable concurrency control. In this approach, programmers are permitted to write, modify and schedule their transactions independent of each other. We have defined a new type of transaction syntax called *compound transactions* and its associated schedules called *generalized setwise serializable schedules*. The classical single level transaction and nested transactions are special cases of compound transactions. Serializable schedules are special cases of generalized setwise serializable schedules. We have shown that generalized setwise serializable schedules are consistent, correct and modular.

In addition, generalized setwise serializable schedules form a complete class within the set of modular schedules. This means that for any given modular scheduling rule R, there exists a generalized setwise serializable scheduling rule which is at least as concurrent as R. Hence, users who are interested in providing a high degree of system concurrency need look no further than generalized setwise serializable schedules. An important special case of generalized setwise serializable scheduling rules is the setwise serializable scheduling rule. We have shown that the setwise serializable scheduling rule is optimal in the set of all application independent scheduling rules. This rule can be "mechanically" applied to schedule any transaction without knowing its semantics. These optimality results are proven under the assumption that the only primitive steps used in the transaction syntax are "read" and "write". The concurrency of (generalized) setwise serializable schedules can be improved by developing families of commutative steps appropriate to one's application. This can be done in a way similar to that done by Korth [Korth 83] for serializable schedules.

Finally, an important issue mentioned but not addressed in this paper is the principle of designing the consistency constraints for the database embedded in a distributed operating system. Generally speaking, using a set of consistency constraints that is weaker than the corresponding ones in the centralized operating

system permits a higher degree of concurrency. However, once consistency constraints are weakened, the complexity of transactions will be increased. We believe that the study of the principles of designing the consistency constraints for a distributed operating system in general and the evaluation of the trade-offs between system concurrency and transaction complexity in particular is an exciting new area of research.

3.4 Distributed Cooperating Processes and Transactions

3.4.1 Co-operating Processes

3.4.1.1 A New Formulation

The synchronization of co-operating processes is an important aspect of an operating system. When the processes are physically dispersed, classical centralized techniques are usually not cost-effective. Our model of data consistency (unlike the serialization model) is able to handle this because the relationships among distributed co-operating processes are represented as partially dependent relations among the state variables of co-operating processes. The synchronization of co-operating processes is thus defined as the maintenance of these dependency relations.

According to this model, co-operating processes generally have two phases — an autonomous phase and a dependent phase. In the autonomous phase, the state variables of the co-operating processes take on values that belong to the set of the cartesian products of the subsets of the domains of these state variables. For example, let the domains of the state variables of processes P_1 and P_2 both be {0,1,2,3}, and let the relation between them be { {0, 1} x {0, 1}, <2, 2>, <3, 3> }. That is, processes P_1 and P_2 can change their states autonomously, as long as their state variables on values from the set of the cartesian products { {0, 1} x {0, 1} }.

In the dependent phase, all state variables in a process must take on values according to the data invariants – e.g., the state variables of P_1 and P_2 above must both have values of either 2 or 3. The problem of ensuring that a set of processes, e.g., P_1 and P_2, will enter their dependent (e.g., identical) states is a matter of maintaining the data invariants "$P_1 = P_2$, $2 \leq P_1, P_2 \leq 3$". This can be done by requiring that the manipulation of the state variables of processes P_1 and P_2 satisfy the conformity condition.

In the autonomous phase, there are no data invariants among the state variables of processes P_1 and P_2 to be maintained, thus it is possible to allow these two processes to maintain a probabilistic relationship among their state variables. This can be accomplished by assigning a joint probability distribution over the set of cartesian products of the processes' state variables. From this joint distribution, we can derive conditional distributions to interpret the probabilistic relationships among the states of processes co-operating in the

autonomous phase. In practice, one often designs a probabilistic algorithm, observes the induced probability distribution, and iterates on the design until the resulting distribution is satisfactory. For example, we can have the following conditional distributions regarding processes P_1 and P_2.

$$P[p_2=0 \mid p_1=0] = 0.8,\ P[p_2=0 \mid p_1=1] = 0.2,$$
$$P[p_2=1 \mid p_1=0] = 0.2,\ P[p_2=1 \mid p_1=1] = 0.8$$

This can be interpreted as P_1 requesting P_2 to be in the same state as P_1, and although P_2 is not obligated to honor P_1's request, P_2 does give P_1's request favorable consideration. Therefore, when P_1 is in state 0 (or 1), P_2 is likely to be in state 0 (or 1).

The need for probabilistic co-operation often arises due to the communication delays in physically dispersed systems. It may be less expensive to maintain certain relationships among data objects indeterministically and recover when necessary, than to force those relationships to always be deterministic.

We now turn to the subject of phase transitions. The transition from the autonomous phase to the dependent phase requires the establishment of a dependency relationship among state variables. Since dependency relationships are defined on version numbers, their establishment includes equalizing the version numbers of each state variable (for instance, by resetting them to zero), and assigning appropriate values to the state variables. In general, a state transition is carried out in three stages. First, if there is more than one process requesting that the transition be made, one of the requesting processes is selected. Next, all of the co-operating processes must be instructed to complete (or abort) any current outstanding autonomous manipulation of state variables, and not to initiate further autonomous manipulation. Finally, values must be assigned to each of the state variables according to the selected processes' requirements, and the version numbers of the state variables must be reset.

The transition of processes from the dependent phase to the autonomous phase is a simple matter. Once a process obtains the right to manipulate the current version of the atomic data set, it can bring the co-operating processes to an autonomous phase by assigning appropriate values from the set of cartesian products to the state variables.

Although there are many different algorithms to implement process phase transition and synchronization activities, we have found that (in a variety of applications) the use of a synchronization path is a effective technique. In the example above, processes P_1 and P_2 co-operate probabilistically in states 0 and 1. Suppose now that P_1 wants P_2 to jointly enter state 2, while P_2 wants P_1 to jointly enter state 3. To resolve such a conflict, a synchronization path could be defined as follows. Any request for dependent co-operation must first be submitted to P_1. If more than one request is received at P_1, one will be honored and forwarded to P_2 where it will also be honored. Requests that were not selected by P_1 will be queued to be selected at later times. The following example illustrates the use of synchronization paths.

3.4.1.2 Example: Remote Process Interruption and Abortion

This example arose in the context of the Spice graphic package, Canvas [Ball 82], which consists of two co-operating processes running on the Accent network operating system. One process is a remote server while the other is a user interface process. The user interface is local to the user's machine and relays user commands to the remote server via messages. For our discussion, we abstract the user interface into four basic commands: EXECUTE, INTERRUPT, CONTINUE and ABORT.

The two basic requirements for this task are: first, it is desirable to minimize message traffic between the two processes; and second, the results of remote service can not be made permanent until the user is informed that the job is done -- that is, the user is given a chance to abort or interrupt the remote process up until the point where he is notified that the job is done. From an implementation point of view, this requirement implies that the user's request should take precedence when there is a conflict between a remote server that is trying to make a result permanent, and a user who is trying to abort (or interrupt) an outstanding server process.

Initially, a remote procedure call based solution was considered because, intuitively, tasks with a remote server seemed to fit this paradigm well. However, it was soon discovered that the conflict between the server process and the user made the remote procedure call approach difficult to use. This is because in a remote procedure call environment, control is passed from the requesting process when the server process is called, and is returned when the server has completed processing the request (or the system detects that the server has failed). The concept of asynchronously interrupting an executing server process is counter to the remote procedure call paradigm. Thus, the problem defined above cannot be easily solved with a classical remote procedure call approach. In this example, the initial attempt to use remote procedure calls resulted in an overly complex implementation. Furthermore, a remote procedure call approach also generates more message traffic, as all inquiries must be forwarded to the remote server for a response, due to the fact that the state of the remote server changes asynchronously with respect to the state of the user server process.

In general the remote procedure call paradigm is appropriate for tasks with master/slave (i.e., hierarchical) control structures, but it becomes much less so for peer processes having symmetrical control relationships. An approach based on our model does not impose such a restrictive control structure on the co-operating processes, and permits the use of local information to reduce the communication overhead. Let the state variable of the interface process be S_u and the state variable of the remote server be S_g. If we maintain data invariants in the form of "$S_u = S_g$", the user interface process can provide the user rapid response by looking only at its local state variable S_u. To ensure that the user-issued ABORT and INTERRUPT commands win any conflicts, we define a synchronization path such that any command must first update the the state variable of the user interface process.

The basic states of the remote server and the user interface processes are called IDLE, SUSPENDED, and EXECUTION, and are labeled as state zero, one and two respectively. The state diagram in Figure 3-1 indicates the defined state transitions, and other command occurrances not defined there will have no effect.

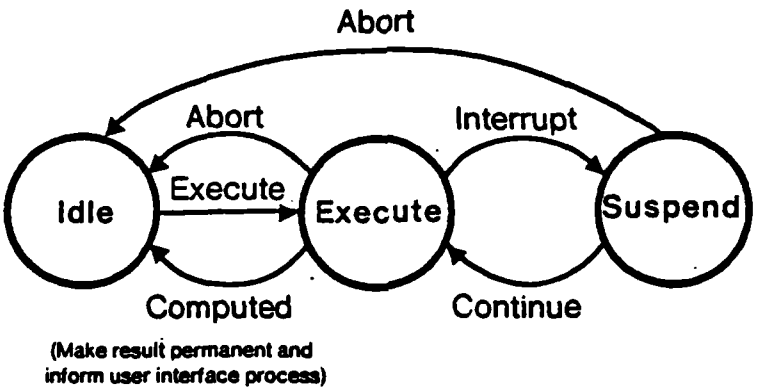


Figure 3-1: State transition diagram of the remote server and the user interface processes

When the system is initialized, $S_u[0] = S_g[0] = 0$. Then, when a user issues an EXECUTE command, S_u will be updated first and $S_u[1] = 2$. The EXECUTE command updates $S_u[0]$, and also updates $S_g[0]$ (via messages), resulting in $S_g[1] = 2$. That is, both the virtual and remote servers go to the Execute state. Suppose that suddenly a user discovers that something is wrong and he issues an ABORT command, while at the same time the server issues a COMPUTED signal (indicating that the computation is done and the result is ready to be made permanent). At this point, there is a conflict between the COMPUTED signal and the ABORT command. Since S_u must be updated first and is local to the user, the ABORT command is applied to $S_u[1]$ first, making $S_u[2] = 0$. When the COMPUTED signal reaches the user interface it will find that S_u is in the Idle state, and will have no effect. On the other hand, the ABORT command, after updating $S_u[1]$, will update $S_g[1]$ and cause $S_g[2] = 0$. Therefore, the ABORT command wins the conflict, resulting in the system returning to the Idle state. Suppose now that the user accidentally issues an INTERRUPT command. The interface process would check its state variable and find that $S_u[2] = 0$. Thus, the INTERRUPT command would be considered invalid and the interface process would warn the user based on its local information alone, and the server would not be affected. Thus, traffic is minimized; there will be no messages between the two processes unless they bring about the state transitions.

3.4.1.3 Example: Process Creation and Destruction

This example arose from the Spice remote file server [Schaffer 82] running on the Accent network operating system, with Unix as the local host operating system. The basic structure of the remote file server consists of a parent process and a set of child processes created to handle users' file manipulation messages. A child process maintains a data port for each of the opened files. The maximum number of such ports that can be supported by a child process is twenty, due to the limitation of Unix on the maximum number of open files a process may have. When a user first sends a request to open a file, a child process will be created for him. When the user wants to open more than twenty files, an additional child processes will be created for him. A child process should be destroyed when it has closed all its ports.

Since the creation and destruction of a child process is a function of the number of ports, the parent process must keep a record of the number of ports that each child process currently has. Thus, let C1.n be a local variable which counts the number of ports at child C1, and let P1.n be the parent's local variable which indicates the number of ports in C1. The standard solution is to construct an atomic data set consisting of {C1.n, P1.n}, with the data invariant "C1.n = P1.n". This data invariant can be maintained by requiring all conflicting transactions to be mutually exclusive with respect to the version numbers of the data objects.

However, there is a problem with this standard solution: it keeps a parent's record consistent with the actual number of ports at the child process for all the values. The OPEN FILE and CLOSE FILE command pair associated with each accessed file causes the number of ports at the child process to be incremented and decremented. This results in two sets of conformal operations to update a parent's record. There is one parent process for many children, and the creation and destruction of ports occurs frequently, so the number of conformal operations needed tends to be large. Thus, the parent process becomes a performance bottle-neck.

This raises the question of whether P1.n has to equal C1.n at all times and for all values. In fact, most of the message traffic is generated to maintain a non-critical relation that could be more efficiently maintained probabilistically. Note that there are only two important values of the port-count, zero and twenty. A port count of zero requires the destruction of the child process, while a count of twenty requires the creation of a new child when a user wants to open more files. Furthermore, we only need P1.n equal to C1.n with some probability when the port-count is twenty. If the parent underestimates the number of ports, additional open file requests will be sent to the child process. However, the child process can return the requests to the parent saying that he already has twenty ports. If the parent overestimates the number of ports, a new child might be unnecessarily created. The time and resources required for that are acceptable in this application. In particular, the probability of creating unnecessary child processes is small, because most users need less than twenty ports.

A port-count of zero, however, is critical because serious abnormalities could occur as a result of the premature destruction of a child process. For example, a child process with ports could be destroyed. Since a child cannot predict the arrival time of a new OPEN FILE command from a user, the child could create a new port after sending a message to the parent process indicating that it has closed all the ports. If a child cannot inform its parent of his status change in time, it could be destroyed by the parent who thinks that the child has no more ports. Note that this problem cannot be solved by letting the parent wait a bit longer after he is informed that the child has no more ports. This is because the arrival time of a new OPEN FILE command from a user is unpredictable. In fact, until the user logs out, the system cannot predict when a user will issue a new OPEN FILE command.

Since a port-count of zero is the only critical value, we can formulate a partial dependency relation as follows. A child and its parent parent process are in an autonomous phase as the port count varies from one to twenty, and they are in a dependent phase when the port count is zero. In addition, when a child has twenty ports, we want its parent process to have a port count of twenty with relatively high probability. This is summarized as:

```
P1.n ▪ C1.n --  with  higher  probability,
     when  the  child  process  enters  or
     leaves  the  state  of  twenty  ports.

P1.n ▪ C1.n --  deterministically,
     when  the  child  process  enters  or
     leaves  the  state  of  zero  ports.
```

This could be implemented by having the child process send a *port-count* message to its parent process when it enters or leaves the state of twenty ports. No effort is made to guarantee that P1.n is equal to C1.n with respect to all concurrent accesses. When the child process enters or leaves the state of zero ports, it initiates a conformal transaction that brings about a phase transition and guarantees P1.n equal to C1.n with respect to all concurrent accesses. When a child has ports between two and nineteen, it will not automatically send any message to its parent because these values are not relevant to the creation or destruction of the child process. However, when a child is interrogated by its parent, it will report its current number of ports via a simple message. This is to permit the operating system to sample the number of opened files for reasons other than process creation and destruction.

By introducing probabilistic co-operation, the communication between the parent and the child for the purpose of process creation and destruction is dramatically reduced. There is essentially one transaction needed during the life time of a child process, independent of the number of files accessed by a user. That transaction is the one that destroys a child process and alters its parent's record. Only in the rare instances when some users need more than twenty outstanding open files are there additional message exchanges

among parents and their child processes. Actual implementation and testing has confirmed that this formulation solves the synchronization problem with a significant improvement in performance (due to the reduced message traffic and message processing time in parent processes).

This example demonstrates that in a message based system the cost of keeping state variables consistent all the time could be high, even on a uni-processor. We believe that in a distributed system the cost of keeping distributed state variables consistent is much higher. Therefore, it is worthwhile to have mechanisms, such as distributed co-operating processes, that permit the separation of the critical parts of relationships that need to be preserved deterministically from the non-critical parts that can be preserved probabilistically.

3.4.2 Co-operating Transactions

3.4.2.1 A New Concept

Atomic transactions are vital to distributed database systems, because they allow the consistency constraints of distributed data objects to be preserved despite the failure of individual pieces of the system. A decentralized global operating system requires the same kind of failure atomicity, and so must be constructed with a transaction facility in its kernel [Jensen 80].

Unfortunately, the serialization model developed for distributed database systems places a fundamental limitation on the use of transactions: i.e., they can model only sequential actions or concurrent actions that are logically equivalent to sequential actions. Yet, a significant part of operating system software takes the form of co-operating processes. The two way communications among co-operating processes make it impossible to transform co-operating processes into co-operating transactions without violating the relative ordering requirement of the serialization model. One of the achievements of our relational model of data consistency is that it provides a foundation for formulating co-operating transactions.

From an application point of view, the need for co-operating transactions arises from the desire to make the actions of co-operating processes atomic. For example, consider the hypothetical case of loan activities within a group of independent banks whose computers are connected by a network. Normally, a bank would handle loan applications by itself; however, if an acceptable loan requires more than 10% of the bank's current capital, the bank must (because of government regulations) ask other banks to syndicate the loan. We can model this as a set of co-operating processes, each of which encapsulates its own confidential financial database. Normally, a process operates in the autonomous phase to handle loan applications by itself. The co-operation starts when a process is asked to join the loan syndication. Once asked, a server will examine its own loan portfolio to determine whether it should accept, refuse, or try to negotiate the terms. Although the formulation of co-operating processes models the loan activity well (i.e., a group of independent processes

who sometimes co-operate), it has a reliability problem. When a computer involved in a syndicated loan crashes, the financial database containing the banking accounts involved in the loan activities might be in an inconsistent state. This is not acceptable, and these process interactions must be made atomic to help eliminate this problem.

Co-operating transactions are transactions that communicate with each other and satisfy the conformity condition. There are two types of data objects manipulated by co-operating transactions. The first is the state variables of co-operating transactions. As with co-operating processes, the partial dependency relations among state variables define the co-operation. The operands of the co-operating transactions are the second type of data object. The manipulation of operands represents the external effects visible to the users. Since operands are organized in the form of disjoint atomic data sets, co-operating transactions can be structured in the form of nested transactions. Each of the sub-transactions of a co-operating transaction operates on one or more atomic data sets and satisfies the conformity condition.

Now we turn to the subject of managing the commit process of a co-operating transaction. A sub-transaction can be committed if and only if the action invariants of both the sub-transaction and all the levels of the co-operating transactions are satisfied. Therefore, an invoked sub-transaction can perform only the first phase of a two phase commit protocol and must leave the final decision of whether to complete or abort the commit to the co-operating transaction. For example, suppose that in the loan syndication problem, bank A originates the loan syndication request, and bank B agrees to participate. The sub-transactions invoked in A and B for handling that loan, such as the transferring M_1 dollars from B to A, and transferring the total amount of M_2 dollars to the customer, must be all done in order to conclude the loan. When all the sub-transactions invoked by A and B have completed their first phase commit, A (the originator of the syndicate) will follow a distributed two phase commit protocol [Bernstein 80] to conclude the loan syndication.

We would like to make two comments on this example. First, the reliability problem *per se* can also be solved by viewing the financial records of each bank as a shared database and using conventional serializable transactions. However, in a typical database approach such as in [Bernstein 80], once an external transaction obtains the write lock, the database is directly manipulated by the transactions. In our approach, external transactions can only indirectly manipulate another bank's financial database via requests to the active local server. It is often important to restrict external users from direct access to another user's (or system) data in order to provide some degree of system security. Secondly, co-operating transactions also provide better concurrency due to the fact that non-serializable concurrent actions are permitted.

3.4.2.2 Example: Graceful Degradation

This example arose from the need to provide a reliable authentication service in the Accent network operating system. Since the database managed by the authentication servers is vital to the integrity of the entire system, it is required that the loss of an individual system element result only in the loss of some performance. Our approach to solving this problem is to use co-operating transactions. The three basic issues in defining co-operating transactions are: 1) the operand atomic data sets; 2) the partial dependency relations among co-operating servers; 3) the definition of sub-transactions. In this case, there are two types of operand atomic data sets. The first is a capability list of users organized as access group lists. The second is records of users' registered ports, which identify processes as having the access rights of their users. The user capability list is partitioned to improve the concurrency of accessing. For reliability reasons, each part of the capability list and the record of a user's registered ports are replicated and distributed in two physically independent machines.

The system authentication servers are organized into a mutual back-up ring. Suppose that there are three servers, S_1, S_2 and S_3, residing on machines one, two and three, respectively. Let the partitioned and duplicated capability lists be $\{L_{1,1}, L_{1,2}\}$, $\{L_{2,2}, L_{2,3}\}$ and $\{L_{3,3}, L_{3,1}\}$, where the first subscript corresponds to the server who is responsible for the set of the two copies of a partitioned list, and the second refers to the location of the host machine. For example, the set $\{L_{2,2}, L_{2,3}\}$ resides on machine two and three, and is maintained by server S_2. A server also has the capability to manipulate the portion of the atomic data sets that resides on his machine, so that it can take over the task of a failed server. For example, server two, in addition to maintaining the set $\{L_{2,2}, L_{2,3}\}$, also takes care of $L_{1,2}$ should server one crash. In addition to the management of the capability list, a server also maintains the records of registered ports. These records are managed in the same way as the capability lists.

The partial dependency relation among servers is as follows. Normally, servers are working independently. Each of them maintains the atomic data sets for which it is responsible. Co-operation among servers is triggered by the events representing the failure or recovery of a server. In Accent, the interprocess communication sub-system automatically monitors, and polls if necessary, each process. Once the failure of a process is detected, the interprocess communication facility will inform the relevant parties. The neighbors of a failed server will co-operatively close the mutual back-up ring. For example, if S_2 crashes, S_1 will recover the atomic data set (such as $L_{2,3}$) by getting copies from S_3. Furthermore, S_1 will ask S_3 to recreate lost redundant files (such as $L_{1,2}$) on machine three. The co-operation associated with the closing of the ring completes when all the relevant atomic data sets are reconstructed. From that point on, S_1 (or S_3) will then manage the atomic data sets that were managed by S_2. When a server process recovers, it will inform its neighbors to transfer the updated atomic data sets back to it. When all the file transfers are done, the recovered server resumes its duty.

The definition of the sub-transactions for this example is straightforward. A sub-transaction is needed to manage the capability list, another is needed to manage records of registered ports, and a final one is needed to perform file management. The first two sub-transactions are used in normal operations, while the file management sub-transactions are used in reconstructing the atomic data sets during the failure and recovery procedures of a server. The action invariants at the server level (i.e., the co-operating transaction level) is simply that all invoked sub-transactions for a task must be all done. For example, when a recovered server is inserted back into the ring, there are two file transfer sub-transactions transferring files back to the recovered one from its two neighbors which must all be completed in order to conclude the insertion.

3.4.2.3 Example: Distributed Load Leveling

This example was conceived to illustrate the communications involved in, and the probabilistic behavior of, co-operating transactions. In this example we examine the problem of distributed load leveling for a point-to-point computer network. In any load leveling scheme, there are two major problems that must be addressed -- the first is providing atomic transfer of work items between work queues, and the second is ensuring the stability of the load leveling operation. The atomicity requirement arises from the need to guarantee that work items will not be lost or duplicated should a node crash during an instance of load leveling. Instability may result from the lack of co-operation among load leveling activities. For example, a pair of heavily loaded nodes (nodes A and B) share a common, lightly loaded neighbor (node C). Nodes A and B might simultaneously observe that node C is lightly loaded, and attempt to off-load some of their work onto it. This would result in node C becoming heavily loaded, and it may then choose to redistribute its load with nodes A and B. This could clearly result in a pathological condition in which work items are repeatedly redistributed.

Thus, for distributed load leveling, it is necessary to have both atomicity of work item transfers, and a form of demand-driven co-operation that is able to adapt to a changing environment. The co-operating transaction paradigm is a formalism that provides a method of meeting these requirements, while permitting highly concurrent execution of the nodes' load leveling functions. The demand-driven, adaptive co-operation between transactions may be represented by probabilistic relations among the state variables of the transactions. The co-operating transaction responsible for load leveling at each node typically operates in an autonomous fashion managing the node's work queue and exchanging load information with other nodes. At some point in time, a node may decide that it is in the best interest of the system to engage in an instance of load leveling. A node would then attempt to enter into a co-operative state with some of its nearest neighbors. This phase of the load leveling function is probabilistic in as far as the neighboring nodes are not constrained to enter into a co-operative state whenever requested to do so. This is because the load information at each node is partial and inaccurate. In the event that none of the neighboring nodes agree to enter into co-operation with the requesting node, the request must be withdrawn and (possibly) reattempted at a later point in time. On the other hand, should a node be successful in entering a co-operative state with one or more of its neighbors, the

group of co-operating nodes collectively enter into a negotiation phase in which it is determined how the load associated with the group should be distributed in order to best accomplish load leveling. It should be noted that the group of nodes involved in co-operation with the node initiating the load leveling attempt could extend beyond its nearest neighbors if a non-neighboring node simultaneously entered into a co-operative state with a common neighboring node.

In general, nodes in the co-operating group will carry out decisions that result from the negotiation within the group. However, due to the dynamic nature of local work item generation and consumption, a node's load could be substantially different at the time a load transfer is attempted from when the group plan was devised. It is therefore desirable for the system to permit local adjustment to the group plan whenever the situation warrants. Allowing local adjustment is another example of the probabilistic co-operation in this example, in that there is no absolute guarantee that the original load leveling scheme will be carried out as planned. For example, the original group plan might require that node A transfer ten work items to node B. However, before the transfer is complete, node B receives a block of locally generated work items. In this situation it may be subsequently determined that the interests of the system are best served by transfering only five of the ten work items. An advantage of using co-operating transactions in such a case is that they permit co-operation (communication) during the execution of the transactions, and thus are able to adapt to environments that change quickly with respect to their execution.

In the co-operating transaction formulation, each node's work queue represents an operand atomic data set which is encapsulated by the co-operating transaction that implements a node's load leveling function. The basic sub-transactions involved in the manipulation of nodes' work queues are ADD, DELETE, and TRANSFER. The ADD and DELETE sub-transactions are used to atomically insert and delete items from local work queues. The TRANSFER sub-transaction carries out specified transfers of work items by invoking the destination node's ADD sub-transaction, sending the work items, and invoking the source node's DELETE sub-transaction. The action invariant of the TRANSFER sub-transaction is that both the remove and insert operations must be successfully completed. Since job queues are encapsulated locally, when the sender's transfer sub-transaction attempts to invoke the receiver's ADD sub-transaction, the receiver may modify the parameters of the ADD sub-transaction. In the above example, ten jobs are sent from node A to node B, however node B takes five work items, instead of ten, and informs node A accordingly. Although node A can reject B's modification by aborting the transaction, A may well re-execute the local ADD sub-transaction and commit the modified transfer. This is clearly more efficient than blindly carrying out the original plan and having to remedy it later.

Finally, it should be noted that the atomicity of work item transfers *per se* can be solved by using a typical database approach based on the serialization model. However, this would be done at the cost of concurrency,

protection, and performance. The loss of concurrency is due to the relative ordering requirement imposed by the serialization model, which is unnecessary for work item transfers. In the co-operating transaction formulation, the relationships between any two work queues are autonomous. The integrity of work item transfers are represented by the action invariant "both the destination node's ADD sub-transaction and the source node's DELETE sub-transaction must be done or neither is done". The work item transfer sub-transactions can be done in any relative order, and may or may not be serializable. The degree of protection is reduced because, with co-operating transactions, each work queue is encapsulated by a local load leveling transaction which controls access to, and maintains the consistency of, the queues. In a typical database approach, one's own work queue may be arbitrarily manipulated by any transaction that obtains a write lock. Finally, performance is sacrificed with serializable transactions due to the fact that they are not able to adapt to a changing environment, as can co-operating transactions which may communicate in the course of their operation.

3.4.3 Conclusion

Our initial experiments with applying these ideas to distributed operating systems have been very encouraging. We believe they are valuable in network operating systems but essential in a decentralized operating system such as ArchOS [Jensen 82] will be. The kinds of interaction amenable to our approach to co-operating processes and transactions are not yet delineated. Neither is it yet very clear what all the implications of these concepts could be on suitable operating system structures. Our research and experiments are continuing, and will be reported in the literature.

3.5 References

[Aho 83] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.
Data Structures and Algorithms.
Addison-Wesley Publishing Company, 1983.

[Allchin 82] Allchin, James E. and Martin S. McKendry.
Object Based Synchronization and Recovery.
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Ball 81] Ball, J. Eugene, Mario R. Barbacci, Scott E. Flman, Samuel P. Harbison, Peter G. Hibbard, Richard F. Rashid, George G. Robertson, and Guy L. Steele, Jr.
The Spice Project.
In *Computer Science Resarch Review*, pages 1-36. Department of Computer Science, Carnegie-Mellon University, 1981.

[Ball 82] Ball, J. Eugene.
Canvas -- The Spice Graphic Package.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1982.

[Beeri 83] Beeri, C., P. A. Bernstein, N. Goodman, and M. Y. Lai.
A Concurrency Control Theory for Nested Transactions.
ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1983.

[Bernstein 79] Bernstein, Philip A., David W. Shipman, and Wing S. Wong.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering : pages 203 - 216, 1979.

[Bernstein 80] Bernstein, Phillip A. and Nathanial Goodman.
Fundamental Algorithms for Concurrent Control in Distributed Database Systems.
Technical Report CCA-08-05, Computer Corporation of America, February, 1980.

[Eswaren 76] Eswaren, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, 1976.

[Gray 81] Gray, J.
The Transaction Concept: Virtues and Limitations.
Proc. 7th International Conf. on Very Large Database, 1981.

[Habermann 76] Habermann, A. N.
Introduction to Operating System Design.
Science Research Associates, Inc., 1976.

[Habermann 79] Habermann, A. Nico.
Implementation of Regular Path Expressions.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1979.

[Hoare 74] Hoare, C. A. R.
Monitors: An Operating System Structure Concept.
CACM, Oct. 1974.

[Jensen 80] Jensen, E. Douglas.
Distributed Computer Systems.
In *Computer Science Research Review*, pages 53-63. Carnegie-Mellon University, 1980.

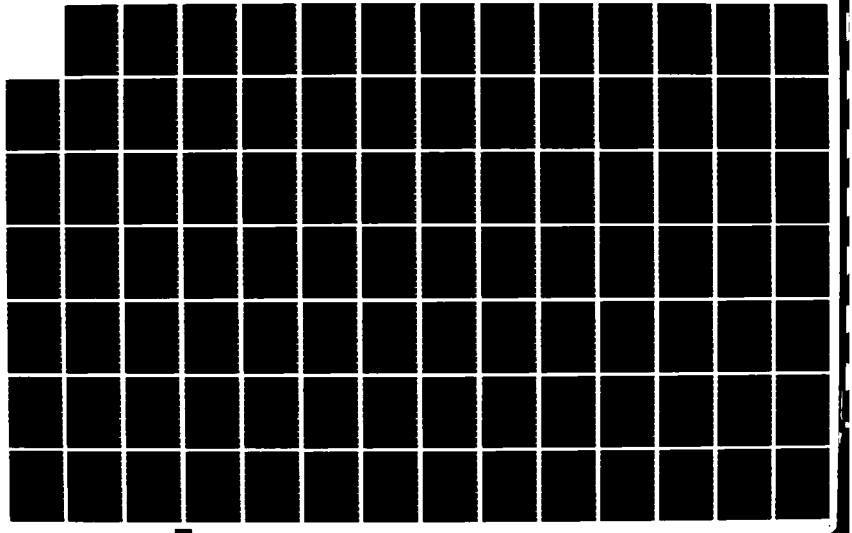
[Jensen 82] Jensen, E. Douglas.
Decentralized Executive Control of Computers.
In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 31-35. IEEE, October, 1982.

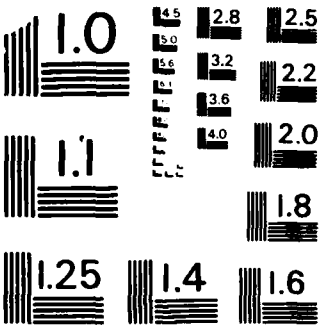
[Jensen 83] Jensen, E. Douglas.
The Archons Project: An Overview.
Proc. International Symposium on Synchronization, Control, and Communication in Distributed Systems, Academic Press, 1983.

[Korth 83] Korth, H. F.
Locking Primitives in a Database System.
JACM, Vol. 30, No. 1, January, 1983.

[Kung 79a] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

AD-A169 754 DECENTRALIZED SYSTEM CONTROL(U) CARNEGIE-MELLON UNIV 2/4
PITTSBURGH PA DEPT OF COMPUTER SCIENCE
E D JENSEN ET AL. APR 86 CMU-CS-ARCHONS-83-1
UNCLASSIFIED RADC-TR-85-199 F30602-81-C-0297 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

[Kung 79b] Kung, H. T. and Phillip L. Lehman.
A Concurrent Database Problem: Binary Search Trees.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1979.

[Lamport 76] Lamport, Leslie.
Towards a Theory of Correctness for Multi-user Database Systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, Inc., October, 1976.

[Lynch 83a] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4, December, 1983.

[Lynch 83b] Lynch, N. A.
Concurrency Control for Resilient Nested Transactions.
Proc. 2nd SIGACT-SIGMOD Conf. on Principle of Database Systems, 1983.

[Molina 83] Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2, June, 1983.

[Moss 81] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April, 1981.

[Papadimitriou 77]
Papadimitriou, C. H., Phillip A. Bernstein and James B. Rothnie, Jr.
Some Computational Problems Related to Database Concurrency Control.
In *Proceedings of the Conference on Theorectical Computer Science.* August, 1977.

[Rashid 81] Rashid, Richard F. and George G. Roberston.
Accent: A Communication Oriented Network Operating System Kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Reed 79] Reed, David P. and Rajendra K. Kanodia.
Synchronization with Eventcounts and Sequencers.
Communications of the ACM 22(2):115-123, February, 1979.

[Schaffer 82] Schaffer, Alex.
LUCIFER: A Mechanism for Transparent File Access in A Unix Network.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1982.

[Schwarz 82] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, 1982.

[Sha 83] Sha, L., Jensen E. D, Rashid, R. F., and Northcutt, J. D.
Distributed Co-operating Processes and Transactions.
Proceedings of ACM SIGCOMM symposium, 1983.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
Journal of the Association of Computing Machinery 27(1), January, 1980.

[Thomas 79] Thomas, Robert H.
A Majority Consensus Apagesroach to Concurrency Control for Multiple Copy Database.
ACM Transactions on Data Base Systems 4(2):180-209, June, 1979.

4. Interprocess Communication

4.1 Overview

Interprocess communication (IPC) is vital to performing decentralized computations. We have not taken the usual approach in distributed systems of simply designing *a* facility for IPC. Our research objectives demand that, as our understanding of decentralized computations and operating systems grows, we must be able to change the IPC facility quickly and easily to provide appropriate support. We are pursuing the use of a technique called "policy/mechanism" separation in the design and implementation of IPC facilities.

Briefly, a *policy* is defined as a specification of the manner in which a set of resources are managed, and a *mechanism* is defined as the means by which policies are carried out [Brinch Hansen 70]. Policy/mechanism separation is a structuring methodology that segregates policies that dictate resource management strategies from mechanisms that implement the lower-level tactics of resource management. Policy/mechanism separation can be applied to a system constructed in a layered fashion; the facility provided at a given level may be implemented by a policy in terms of mechanisms, and that facility may in turn be used to create mechanisms at the next higher level.

The design and implementation of IPC facilities are an important part of multiprogramming systems in general, and is critical to "distributed systems". Furthermore, because IPC facilities have great impact on the systems of which they are a part, serious thought must be put into their functionality and structure.

Policy/mechanism separation has been shown to be valuable in the design of general operating system facilities [Brinch Hansen 70, Kahn 81, Wulf 74], but primary emphasis has been on the area of process scheduling [Bernstein 71, Levin 75]. Furthermore, until now there have been no explicit attempts at applying these principles specifically to IPC facilities. There is reason to believe that policy/mechanism separation is likely to prove useful in achieving a number of goals for IPC facilities, such as:

- the flexibility to create a wide range of different facilities,
- support for multiple, different, coexistent IPC facilities, and a
- viable approach to providing hardware support for IPC.

This research will result in *a set of IPC mechanisms* that will support the implementation of a wide range of IPC facilities. Another contribution will consist of an *evaluation of the policy/mechanism approach to IPC*, based on implementations of the previously specified mechanisms and a chosen set of IPC policies. Further contributions of this research will include a *taxonomy* of the IPC design space, and a *logical framework* to

represent various implementations of a range of IPC facilities, an evaluation of the degree to which *multiple IPC facilities* can be simultaneously supported, and whether the set of proposed IPC mechanisms can be effectively *supported with hardware* (which would include descriptions of proposed hardware mechanisms).

Although there has been a great deal of work in the general area of IPC [Northcutt 83], relatively little of that work is strongly related to the research outlined in this document. Of the many different types of IPC facilities that exist or have been proposed, few have had flexibility (in the sense of permitting a range of different facilities) as a goal, although some contend that their system is capable of implementing a wide range of IPC facilities [Rao 80]. Furthermore, while others have attempted to provide hardware support for their particular IPC facility [Cox 81, Ford 77, Giloi 81, Spier 73], such support tends to be unsubstantial and highly inflexible. To the best of our knowledge, there are no instances of IPC facilities explicitly designed and implemented according to the principles of policy/mechanism separation. This is despite the fact that some IPC facilities consist of operations known as "primitives" [Liskov 79].

4.2 The Separation of Policy and Mechanism in IPC

4.2.1 Introduction

This research explores the separation of policy and mechanism in the design and implementation of interprocess communication (IPC) facilities. Briefly, a *policy* is defined as a specification of the manner in which a set of resources are to be managed, and a *mechanism* is defined as the means by which policies are carried out [Brinch Hansen 70]. Policy/mechanism separation is a structuring methodology that segregates policies that dictate resource management strategies from mechanisms that implement the low-level tactics of resource management. This technique has been suggested for, and applied to, the design and implementation of general operating system facilities [Levin 75]. Policy/mechanism separation can be applied to a system constructed in a layered fashion; the facility provided at a given level may be implemented by a policy in terms of mechanisms, and that facility may in turn be used to construct mechanisms for a facility at the next higher level.

The design and implementation of IPC facilities are an important part of multiprogramming systems in general, and are critical to *distributed systems*[1]. In systems whose software is constructed as a collection of conceptually distinct programming elements (e.g., processes), an IPC facility is the fundamental means by which the components of the system communicate with one another (and in some cases with other system facilities). In general, IPC facilities have a great impact on the nature of systems they are a part of; a great

[1] We use this term in the popular sense, i.e., meaning any system with more than one processor.

deal of effort should therefore be put into the design and implementation of these facilities. Not only does the logical functionality of IPC facilities affect the structure and behavior of the systems, but the nature of systems themselves affects the requirements for their IPC facilities. Furthermore, both the degree to which a system places demands on an IPC facility and the response time constraints of a system influence the efficiency requirements of a system's IPC facility.

Policy/mechanism separation has been shown to be valuable in the design of general operating system facilities [Brinch Hansen 70, Cox 81, Wulf 74]. At this time, however, there have been no explicit attempts at applying these principles specifically to IPC facilities; the primary emphasis has been in the area of process scheduling. Nonetheless, there is reason to believe that a policy/mechanism separation approach could prove useful in achieving a number of goals for IPC facilities.

The benefits of an IPC facility based on policy/mechanism separation can be expected to include the following:

- providing an IPC facility flexible enough to permit the creation of a wide range of different IPC facilities through the application of various policies to IPC mechanisms;
- supporting multiple, different "native" IPC facilities that can simultaneously coexist at the same level in a given system; and
- an approach to providing hardware support for IPC facilities, to improve performance without sacrificing flexibility.

Separating policy from mechanism yields primitive functions (mechanisms) with which various IPC facilities can be implemented by changing policies (i.e., many specialized IPC facilities can be implemented in terms of a single set of mechanisms). Policy/mechanism separation thus results in highly flexible IPC facilities. This property is particularly useful for a testbed system, designed for experimentation with various (possibly unforeseen) operating system structures.

It should be noted that the separation of policy and mechanism in the design and implementation of IPC facilities is not necessarily being suggested as a general approach to the construction of IPC facilities. Rather, IPC facilities designed according to the policy/mechanism separation approach offer certain (unique) benefits, and lend themselves best to certain specific environments. It may be the case, however, that this approach to IPC facility design is sufficiently broad in its applicability that it might be used more generally. Such an occurrence would be similar to cases where writeable control store was provided in a prototype computer architecture for development purposes, but proved to be so useful that it was included in later production versions of the computer.

4.2.2 Background

Before we can further discuss the separation of policy and mechanism in IPC, it is important that the notion of IPC be somewhat better defined, and our scope of interest in IPC more clearly delineated. In addition, the terms *policy* and *mechanism* must be defined along with the general concept of *policy/mechanism separation.*

4.2.2.1 Definition of Interprocess Communication

A common method of structuring a programming system is to construct it from a (possibly hierarchically structured) collection of program entities. These entities are commonly known as *processes*, an operating system supported abstraction that can be thought of as the basic unit of computation and concurrency in modular programming systems [Habermann 76]. For the purposes of this discussion, we consider a process to be a unit of computation that is serially executed on an underlying (real or virtual) machine, in (real or virtual) asynchronous concurrency with respect to other processes. Despite the abundance of more formal definitions of processes, there is not one commonly agreed upon nor more appropriate for our immediate needs.

To cooperate, processes require some means of communication. This communication can take many forms, but IPC is the activity of deliberately and explicitly exchanging information among processes. In the case where the processes wishing to communicate have intersecting domains, IPC can be carried out by one process instantiating the information to be exchanged in a shared portion of the processes' domains; communication then largely consists of coordinating access to the shared information. Where process domains are disjoint, IPC is performed by moving information from the domain of one process to the domain(s) of one or more other processes. Throughout this research we will consider only the latter case of IPC, and communication among processes by such means as shared memory, common files, etc. is not included here.

At the next lower level of detail, IPC can be roughly thought of as being composed of four basic activities:

1. the specification of the participants involved in a given instance of IPC (i.e., which processes are involved and in what capacities, e.g., message source, message destination, etc);

2. the instantiation of information, initially local to one process, in the domain of one or more other processes (i.e., *what* information is to be exchanged and *how* the exchange is to be carried out, e.g., reliably, sequenced, broadcast, multicast, etc.);

3. the act of causing, detecting, or being made aware of, various events involved in the coordination of communication activities (i.e., how are the participants able to create and detect events such as "message queued at destination process", "message accepted by communication subsystem", etc.);

4. the interpretation of (at least portions of) the information instantiated in a process' domain as a result of an act of IPC (i.e., to what extent the entire message is to be decoded by the processes and the system).

Each of these fundamental activities must be performed explicitly or implicitly in an instance of IPC, and an IPC facility must provide all of the functions to do so. Furthermore, there exist a great many ways in which these basic activities can be provided to a user process, and different IPC facilities provide them in different forms.

4.2.2.1.1 The Role of IPC in Programming Systems

A common form of structured system design and implementation is known as *layering* [Dijkstra 68]. A layered system creates successively higher levels of functionality by implementing each layer in terms of the underlying layers[2]. In such a system, the peer processes at each layer require a form of IPC (known as protocols [Zimmermann 80]) among themselves. The IPC facilities used by each layer could be the identical, basic IPC service, or each layer could make use of a different IPC facility with increasing functionality. For example, in the RIG system a fundamental form of IPC is used to provide access to more elaborate forms [Lantz 80]. Thus the IPC facility in a system could also be layered -- each layer of IPC could be increasingly rich in functionality, and the IPC service at each layer could be well suited to the type of communication that occurs among processes at that specific level.

In a layered system, a form of communication is required for higher layers to invoke functions provided by lower layers. This is usually known as an *interface*, i.e., communication among processes in different layers [Zimmermann 80]. While operating system processes at layers above an IPC facility can use that facility for both interface and protocol communication, this is not true for processes using the lowest-level (i.e., the fundamental) IPC facility. These processes require some other form of communication (which is not IPC by our definition) to interface to this facility, due to the obvious circularity of needing to use a facility in order to access (or provide) that same facility. There exist a number of different means through which processes can access fundamental IPC facilities, including procedure calls, language constructs, supervisor call-type instructions, etc..

4.2.2.1.2 The IPC Facility Design Space

In addition to the wide variety of options that exist for the implementation of IPC in a system, there are many different types and degrees of functionality that can be provided by an IPC facility. IPC can range from a simple device-like form to a complex, transaction-oriented facility. To illustrate the possible differences in IPC facility design, a number of options are given below. Note that there is no attempt to suggest that these are the most common variations, nor is anything to be inferred from the order in which they appear. Furthermore, it should not be assumed that all of these options are compatible with one another.

- An instance of IPC is usually initiated by one of the processes involved in the communication: the

[2] In a strictly layered system, each layer is defined exclusively in terms of the objects (i.e., data structures and operations on them) provided by the layer immediately below it.

process that generated the information to be transferred (i.e., *source initiated*); the intended recipient of the information transfer (i.e., *destination initiated*); or alternatively a process that is neither source nor destination of the transfer (i.e., *third party initiated*) [Jensen 78a].

- It is necessary that the parties involved in a given act of communication be specified. This can occur by addressing a message with the name(s) of the destination process (i.e., *destination addressing*), the name of the source process (i.e., *source addressing*), or by defining a special "tag" field (i.e., *content addressing*). In addition to these addressing methods, it is possible to perform implicit addressing by the use of *connections*, or by using a logical *service* address. (Note that one or more of these pieces of information could be included within a message, but in this discussion we are referring only to the information used in performing the act of addressing.)

- The actual exchange of units of information (e.g., *messages*) can take place in many different ways. Much like parameters passed in procedure calls, messages could be passed by *value*, *reference*, or *function*. Also, the relationship between sources and destinations of messages could be one-to-one (i.e., *two-party*), one-to-many (i.e., *multicast*), one-to-all (i.e., *broadcast*), or all-to-one (i.e., *promiscuous*). The behavior of messages with respect to their receive semantics could be once-and-only-once, at-least-once, or something different. Message transfers could even be guaranteed to be atomic with respect to other message transfers (i.e., a *transaction*).

- Control information passed to the client of an IPC facility from the facility can be described as being either *imperative* or *interrogative*. In the imperative form, control information is made available without explicit action on the part of the client (e.g., an interrupt, the unblocking of a process, etc.). The interrogative form of control information transfer requires that the client issue a form of "query" operation to obtain the control information.

  The type of information that may be passed in these ways includes the status of ongoing communications (e.g., "message accepted by the local communication subsystem", "message accepted by destination process(es)", etc.), or the state of the communication subsystem (e.g., "path I operational", N messages of type T queued for process P", etc.).

- The processes involved in an instance of IPC can have a number of different relationships among themselves, with respect to their control flow. There might be no synchronization between the source and destination processes (i.e., *asynchronous communication*), either the source or the destination process could suspend execution until the other has executed a send or receive (i.e., *semi-synchronous communication*), or both the source and destination processes could suspend until the other issues a matching send or receive command (i.e., *synchronous communication*).

- There must be some degree of agreement on the format of messages in order to ensure that processes can interpret the information exchanged. This implies that the format of messages must either be entirely fixed, or at least a portion that describes the remainder of the message must be fixed. Furthermore, if the communication subsystem must interpret the contents of messages (e.g., to transform local capabilities into their remote manifestations), the message format must accommodate this either by fixed fields or special, reserved markers.

Different IPC facilities can coexist in a single system (e.g., in RIG there is Rashid's IPC along with a variety of facilities based on Xerox protocols [Fleisch 81], and some versions of UNIX[3] have both pipes and Rashid's

[3] UNIX is a registered trademark of Bell Laboratories.

IPC [Rashid 80]). In addition to having a variety of IPC facilities at different layers in a system, it is possible for the same functionality to be implemented at different layers. For the most part, the issues of functionality and layering are independent with respect to IPC facilities; it is typically the case, however, that functionality increases as IPC appears in higher layers in a system.

4.2.2.1.3 Our Scope of Interest in the Universe of IPC Facilities

Out of the universe of possible IPC facilities, we are confining our present interest to a specific subspace. This is intended to restrict the emphasis of our work to the forms of IPC that we consider to be the most appropriate for *distributed systems* in general, and most relevant to our research on the Archons project in particular [Jensen 83]. By restricting our scope of interest in the IPC design space, we are attempting to reduce the number of IPC facilities we must consider by eliminating those facilities which (in our opinion) have undesirable or uninteresting characteristics.

The IPC facilities of greatest interest to us in this research share the following characteristics:

- they are based on message passing (as opposed to procedure calls, etc.);
- communication is primarily via an explicit IPC facility (not shared memory, shared files, I/O, etc.);
- all IPC is performed with the explicit consent of all the communicating processes (not by a unilateral action on the part of some arbitrary process).

4.2.2.2 The Separation of Policy and Mechanism

The concept of policy/mechanism separation was described by Brinch Hansen in 1970 [Brinch Hansen 70] and applied in the RC4000 system [Brinch Hansen 71]. Other notable systems which attempted to separate policy and mechanism within their operating systems include the Hydra/C.mmp system [Wulf 74] and the iAPX 432/iMAX[4] system [Kahn 81]. Experience has shown policy/mechanism separation to yield a number of benefits in the design and implementation of systems.

4.2.2.2.1 Policy/Mechanism Separation as a General Structuring Methodology

There are a number of concepts associated with the design and implementation of computer systems that are (at least superficially) related to the notion of policy/mechanism separation, the most obvious of which is abstraction. Policy/mechanism separation could be thought of as a form of abstraction, in that policies define higher-level functions implemented in terms of lower-level ones (i.e., mechanisms). It is more useful, however, to consider policy/mechanism separation as a technique for implementing a given layer of functionality, which involves partitioning the layer into a part that dictates behavior and a part that carries it out.

[4] iAPX 432 and iMAX are registered trademarks of the Intel Corporation.

Another related concept is that of separating specification from implementation. In a sense, a policy is a specification of a function and the mechanisms used to carry the policy out are its implementation. However, this is best thought of as an issue that is orthogonal to that of policy/mechanism separation: the design and implementation of the policy and mechanism portions of a facility could be performed by separating the specification and implementation of each part.

Information hiding [Parnas 72] is a concept also related to policy/mechanism separation, inasmuch as policies are implemented in terms of mechanisms that serve to isolate the policy maker from the details of the mechanisms' implementations. However, the primary objective of information hiding is to insulate the interface of a facility from internal changes in the facility's implementation. This is as opposed to policy/mechanism separation which attempts to insulate the internal implementation of a facility from changes in its external interface.

According to our interpretation of this concept, we now define some terms and present a simple view of system structure based on the separation of policy and mechanism.

- *Facility*: a service characterized by a collection of operations that comprise its interface. A facility, implemented according to a policy/mechanism separation approach, consists of a collection of mechanisms and a policy which governs the manner in which the mechanisms' constituent primitives are invoked.

- *Policy:* a plan of action relating to the management of a collection of resources, based on "global" objectives, general goals, and acceptable procedures. In facilities implemented according to a policy/mechanism separation approach, policies are carried out by the invocation of primitives.

- *Mechanism*: a related collection of functions that carry out various aspects of a common function. Mechanisms are used to carry out policies in policy/mechanism separation implementations of facilities.

- *Primitive*: a function that carries out a single aspect of particular function. Primitives are the entities which are invoked in order to carry out an operation on behalf of a higher-level entity. A mechanism is composed of primitives that perform related operations.

The conceptual boundary between policy and mechanisms (for a given facility) might be viewed as the separation between a pair of layers in a functionally layered structure. Also, it is clear that facilities that exist at a certain level, may support (or implement) mechanisms at higher layers in a system. However, all discussion of policy/mechanism separation in this document should be assumed to be in the context of a facility within a single layer of a system. Policy/mechanism separation is, in effect, a methodology that guides the implementation of a given layer.

A given facility is implemented by making use of mechanisms according to a given policy. The same mechanism may be used in more than one facility, and a given facility could be implemented using the same

policy but different mechanisms. Also, separate facilities may be simultaneously provided by different policies implemented in terms of the same set of mechanisms. However, arbitrary policies may not be compatible, and as such may not be capable of simultaneously coexisting in a system. On the other hand, the choice of a given mechanism tends to be largely independent of other mechanisms. The choice of mechanisms can affect the types of policies that can be carried out and the cost of carrying out the policies.

An example of a facility at the operating system level is one that permits the multiplexing of a physical processor. A scheduling facility could be implemented according to such policies as: *Round-Robin*, *Shortest-Processing-Time-First*, or *Priority*. Any of these policies might be implemented in terms of the same processor multiplexing mechanism, which could consist of a set of operations such as: "define the selection discipline", "select one of N processes", "stop currently active process", and "start process P". A scheduling facility implemented with such mechanisms might exhibit the characteristics of policy/mechanism separation.

4.2.2.2.2 The Separation of Policy and Mechanism in IPC

Despite the fact that the separation of policy and mechanism has been (more or less) successfully applied to various parts of a number of systems, IPC has not yet received the benefit of such a treatment. To this point, the primary emphasis on applying policy/mechanism separation has been in the area of process scheduling and memory management [Levin 75]. Among the arguments for not applying policy/mechanism separation to IPC facilities might be: due to its complexity, IPC is a facility which does not readily lend itself to such an effort; it does not make sense to separate policy from mechanism in IPC, because it is such a low-level facility that the benefits are outweighed by the costs; or there is nothing to be gained from an endeavor of this sort that couldn't be better accomplished in some other fashion. Each of these objections will be shown to be unreasonable in some cases. Furthermore, a number of counter-arguments can be made which suggest there is value in investigating policy/mechanism separation with respect to IPC.

4.2.2.3 Interprocess Communication for Decentralized Computer Systems

One of the objectives of this research is to determine the degree to which the separation of policy and mechanism will provide an effective methodology for the design and implementation of a flexible IPC facility. This is of special interest to us because a great deal of flexibility is required of an IPC facility for the support of research on decentralized operating systems (DOS's) [Jensen 83]. A DOS is based on the concept of multilateral control, and intended to reside on a physically dispersed computer (e.g., a local network-like architecture), thereby forming a decentralized computer system (DCS) [Davies 81]. DOS design is a new area of very active research and there is very little practical experience; therefore much is to be gained from design and implementation experiments. The most obvious approach to obtaining empirical data on DOS's is to construct a DCS testbed on which prototype DOS's and various DOS concepts can be implemented and evaluated.

In any DOS implementation based on the concept of multiprogramming (or cooperating concurrent programs in general) there must be an IPC facility with which the constituent processes of the DOS communicate. It is clear that the choice of an IPC facility can have a profound influence on the structure of the programming systems that make use of that facility. Therefore, the IPC facility for a DOS testbed system should support a range of software structures that might be used in constructing DOS's. However, the software structures most appropriate for DOS's have, as yet, not been conclusively identified. This suggests that a good IPC facility for a DOS testbed system would be one that permits a wide a range of different IPC facilities (and hence software structures) to be implemented or efficiently emulated.

As a result of research on the fundamentals of DOS design, a few general observations can be made on the implications of DOS's on IPC facilities. It is clear at this point that *master/slave* type relationships will not be the predominant form of process structure, but rather that general non-hierarchical process-process relationships (e.g., collections of negotiating peers) will be most common. This implies that synchronous ***Send&Wait*** or procedure call-oriented IPC facilities will be less appropriate for DOS's than facilities providing message based, ***N-party*** communication transactions. Furthermore, the cooperative nature of the collective decision-making in DOS's suggests a greater amount of system-generated communication and makes a greater demand on the efficiency of the IPC facility (and its underlying implementation) than does a typical local area network operating system.

4.2.3 Rationale

IPC is a highly important operating system facility that greatly influences the structure and performance of systems. A number of different approaches exist for providing an appropriate IPC facility for a given system. Among these approaches are: a facility implemented with a highly parameterized interface, a facility with a strictly layered implementation, and a facility that employs a policy/mechanism separation approach. The policy/mechanism separation approach holds a number of benefits that are not to be found in other approaches. For example, separating policy and mechanism seems to be an exceptionally good method of choosing the hardware/software boundary for IPC facilities, and of determining the appropriate primitives to provide (or support) in hardware. Through the separation of policy and mechanism in IPC, it may also be possible to have different IPC facilities simultaneously coexist in a system. Furthermore, by separating policy and mechanism an IPC facility can be defined that meets the requirements for DOS research, in a manner superior to that which can be accomplished through alternative approaches.

4.2.3.1 Significance of Interprocess Communication Facility Design and Implementation

IPC facilities stand out as special in comparison to other operating system facilities. An IPC facility is typically included in an operating system kernel[5], and recently many of the other (non-kernel) facilities are being made available through a system's IPC facility. The special role that IPC plays in a system clearly sets it apart as a facility on which much of an operating system can be constructed, just as most operating systems in the past were built on memory management. The degree to which a system places demands on an IPC facility (either due to accessing other facilities through IPC, or due to process communication) influences the efficiency requirements of the facility.

The choice of IPC functionality can have a great effect on the structure of the system that makes use of it. For example, the type of IPC facility provided can influence the forms of control structures possible among cooperating process. This can be seen in the case of a system constructed on an IPC facility that provides only synchronous ***Send&Wait*** and ***Receive&Wait*** constructs (similar to remote procedure call semantics). In such a system, processes are constrained to exhibit coroutine-like behavior, where there is only a single point of control at any point in time (thus restricting the potential for concurrent execution). However, it might be possible for a process in such a system to spawn concurrent child processes that could carry out the synchronous IPC concurrently with the execution of the parent process. Thus, either the control relationships between processes are unnecessarily limited, or a potentially large number of child processes must be introduced to simulate the desired behavior, adding not only to overhead but to the overall system complexity. Another example of how IPC functionality can have an effect on systems can be seen in the impact that IPC exception conditions and their side-effects can have on system design. An example of such an effect is the structuring of system service processes and operations to be idempotent in order to cope with extraneous service requests resulting from replication of messages in the IPC facility.

The performance of an IPC facility can also have an effect on system structure because the cost of communication frequently influences the design and partitioning of systems. This is evident in the fact that virtually all distributed system software is partitioned according to minimum communication bandwidth, as opposed to some other metric (such as information hiding [Parnas 72]). Entire software structuring techniques have been developed in response to the relative cost of inter- versus intra-process communication, or the cost of local versus non-local IPC. Examples of such structures include CLU Guardians [Liskov 81], Thoth Pods [Cheriton 79], and StarOS Task Forces [Jones 79].

---

[5] The kernel is the part of the operating system necessary to support basic abstractions such as processes, and mask undesirable portions of the underlying physical hardware.

4.2.3.2 Alternative Approaches to Flexible Interprocess Communication Facilities

An IPC facility for a system where the software structure is not well defined must support a wide range of different types of IPC if the facility is not to adversely impact the design of the software that uses the IPC facility. There exist a number of alternative approaches to achieving such a flexible IPC facility, and the most significant of these are briefly discussed here.

4.2.3.2.1 A "Parameterized" Approach

One approach would be to make some educated guess as to what the range of IPC requirements might be and specify an IPC facility that can meet all of the requirements. This approach is characterized by a heavily parameterized facility interface, whose flexibility derives from the range of functions achievable via this interface. The problems with such an approach include the difficulty of ensuring that all desired facilities can be implemented, and the logical complexity of making use of such a parameterized facility.

An example of such a parameterized interface can be seen in the MU5 compiler target language model (CTL) [Barringer 79]. This model provides an abstract machine suitable for use as an intermediate language interface to a collection of high level languages (e.g., Algol 60, Algol 68, PL/I, Fortran, etc.). The interface provided by CTL was a highly elaborate, parameterized interface that included features specialized for each of the languages to be supported. Experience with the CTL interface showed it to be adequate for generating efficient object code. However, the difficulty of using the highly complex interface and the overall poor performance of the compilers led to the development of a lower level interface which proved easier to implement compilers for, and generated codes were more efficient.

4.2.3.2.2 A "Strictly Layered" Approach

An approach that does not rely on an *a priori* definition of the requirements for an IPC facility involves the choice of "lowest common denominator" type of low-level facility. The problem of IPC flexibility is dealt with by providing the simplest possible IPC facility out of which a range of higher-level facilities can be constructed (through successive layers of virtualization). For example, such a fundamental IPC facility might include as operations a ***Non-Blocking Send*** construct and a ***Wait for Message*** construct, the concatenation of which implements a ***Blocking Send*** construct (at a higher level of abstraction). This reduces the problem of having to predict all possible higher level facilities in an *a priori* fashion by only requiring that the IPC facility designer ensure it is possible to construct the desired facility from the given lower level one. However, this approach provides flexibility at the cost of performance; each successive layer of virtualization exacts a cost, which accumulates across all the layers and negatively affects the performance of the higher-level IPC facilities.

In addition to the performance penalties incurred in a strictly layered approach, it may be quite difficult

(and occasionally impossible) to construct particular functions out of a given set of low-level operations. For example, implementing a ***Selective Receive*** construct in terms of a simple ***Receive*** requires multiple message exchanges and a great deal of logical complexity.

4.2.3.2.3 A "Policy/Mechanism Separation" Approach

The policy/mechanism separation approach provides a method of decoupling the requirements driven portions of a specific IPC facility from the generic mechanisms required for all types of IPC facilities. This technique offers a means of implementing a generic set of communications mechanisms, making it possible to easily implement and modify arbitrary IPC facilities through different organizations of invocations of the mechanisms (i.e., policies). The policy/mechanism separation approach differs from a strictly layered approach in that the requirements driven (hence potentially variable) policy decisions are implemented directly on top of a collection of mechanisms, as opposed to being constructed out of an arbitrary number of successively higher-level facilities. Additionally, while the policy/mechanism separation approach creates a pair of layers (i.e., the policy layer, and the mechanism layer), the policy/mechanism interface does not necessarily provide a complete facility. It is only through the invocation of the mechanisms in accordance with a specified policy that a complete IPC facility can be considered to exist. This is as opposed to a strictly layered approach, which provides a complete (albeit possibly functionally primitive) IPC facility at the lowest level.

While the policy/mechanism separation approach permits the direct implementation of IPC facilities without intervening layers of functionality, this is not to say that the benefits of hierarchically structured function composition cannot be used in the construction of IPC facilities via policy and mechanism separation. Clearly, the more levels of interpretation required to provide a given service, the poorer the performance of the ultimate service will be. This suggests that the policy/mechanism separation approach would provide implementations with better performance characteristics than those based on a strictly layered approach. While this may be the case, it should also be clear that a specialized (and monolithic) implementation of a facility will most always have greater performance than a facility designed to be highly flexible. We explicitly acknowledge this fact, and willingly accept somewhat sub-optimal performance in return for flexibility.

In attempting to provide a flexible IPC facility through a policy/mechanism approach, it is important to define mechanisms to simplify the design and implementation of the policy components of the facilities as much as possible. This can be done at the expense of additional complexity in the mechanism portion, because the cost of the design and implementation of the mechanisms will be non-recurring, while the cost of implementing different policies recurs each time a different IPC facility is created. This implies that the IPC mechanisms' level of functionality should be raised to the greatest extent possible, without having the mechanisms dictate policy in any way. It should be noted that restricting the range of policies a set of mechanisms can carry out, can be thought of as dictating policy.

It is apparent that in such an effort one is faced with a problem analogous to that which is currently at the heart of the instruction set architecture (ISA) debate known loosely as the "RISC/CISC" argument. This problem revolves around the attempt to optimize a number of attributes (such as execution speed, code size, implementation complexity of a vehicle to interpret the ISA, etc.) based on varying the level of functionality of the interface provided by the ISA. The interface provided by a set of IPC mechanisms is subject to an argument similar to one found in the RISC/CISC debate. If high level of functionality mechanisms do not meet the exact needs of a policy implementer, the cost of achieving the desired result may be greater than that incurred using only lower level mechanisms. Clearly, this requires that a great deal of effort be made in determining the optimal level of functionality for a set of mechanisms. The choice of the most appropriate IPC mechanism interface seems somewhat more manageable than the analogous ISA problem; the choice IPC mechanism interfaces is somewhat simplified by the fact that the range of policies to be implemented can be reasonably well defined.

4.2.3.3 Applying Separation of Policy and Mechanism to Interprocess Communication

To this time, there have been no attempts (that we are aware of) to separate policy and mechanism in IPC. This is despite the fact that IPC facilities can be expected to benefit from the application of the concept of policy/mechanism separation in ways similar to those achieved with other operating system facilities.

4.2.3.3.1 Providing Flexible IPC Facilities

Separating policy from mechanism yields a collection of functions (mechanisms) with which various IPC facilities can be implemented by changing policies (i.e., many specialized IPC facilities can be implemented in terms of a single set of mechanisms). Policy/mechanism separation thus results in highly flexible IPC facilities. This property is particularly useful for a testbed system, designed for experimentation with various (possibly unforeseen) operating system structures.

Furthermore, a separation of policy and mechanism in an IPC facility permits the mapping of a set of slightly abstracted, lower-level mechanisms, into the desired higher-level IPC facility. This mapping is performed without the overhead incurred by traditional multi-layered implementations, yet does not abandon the benefits of abstraction. The policy/mechanism separation approach falls between multi-layered implementations (which impose a substantial cost in terms of interfacing overhead), and monolithic implementations (which sacrifice flexibility, modularity, and intellectual manageability).

The separation of policy and mechanism also allows operating system designers to choose IPC policies that are specialized for particular environments, as opposed to having to make the best of whatever facility is provided by the kernel. A user of the IPC facility can, by specifying different IPC policies, create a range of custom IPC facilities. This customization of an IPC facility's interface is done once for each new IPC facility

desired, by the individuals who know the most about a system's requirements. Furthermore, the IPC interface is customized in such a way as not to obscure or restrict the power of the underlying mechanisms, and to insulate the IPC facility designer from the full complexity of the underlying physical resources. These are all considered to be desirable characteristics for operating system facilities [Lampson 83].

4.2.3.3.2 Support for Multiple Coexistent IPC Facilities

In an IPC facility designed according to a policy/mechanism separation approach, multiple types (or versions) of IPC facilities could simultaneously coexist in a system, provided that they do not place conflicting demands on the underlying mechanisms. The policy/mechanism separation approach permits a decoupling of multiple, coexisting IPC facilities that is not possible with other approaches.

The existence of multiple IPC facilities in a system would clearly require that processes use a common IPC facility for each instance of IPC. This may partition the processes in a system into groups according to the IPC facilities that they have access to. Multiple coexistent IPC facilities are currently possible by other means, however in most all extant cases, different IPC facilities are implemented in terms of some other IPC facility (typically in a layered fashion). In a facility implemented according to the policy/mechanism separation philosophy the differing IPC facilities can be directly implemented by different policies, thereby eliminating the potential for problems due to circular requirements (i.e., a facility built in terms of another facility, which is in turn built in terms of the former facility).

4.2.3.3.3 Providing Hardware Support for IPC

An important attribute of IPC facilities (and the one that receives the greatest amount of attention) is performance. This is largely due to the fact that IPC is a fundamental facility on which increasingly more operating systems are relying on to an increasingly greater extent[6]. A common method for enhancing the performance of IPC facilities has been to reduce the amount of interpretation involved in providing the desired facility. We have pointed out that a policy/mechanism separation approach provides a facility on top of a single level of virtualization. If the physical resources of a system directly implement the functionality specified for a set of IPC mechanisms, an IPC facility could be implemented with the least amount of interpretation (and hence, the greatest performance) possible without making the sacrifices associated with monolithic implementations. Thus, a benefit of policy/mechanism separation in IPC is a promising approach for applying inexpensive hardware (in the form of VLSI components) to the problem of providing increased IPC facility performance, without restricting flexibility.

---

[6]The emphasis on performance may also be attributed to the fact that (as in computer architecture) it is significantly easier to derive something that can pass as a measure of performance, than to measure the similarly important attributes of modularity, fault tolerance, life-cycle cost, etc.

4.2.3.3.4 IPC for Decentralized Operating System Research

The property of flexibility, without the typically attendant loss of performance, is a major benefit of IPC facility design and implementation based on policy/mechanism separation. This attribute is extremely useful in an environment such as that of performing empirical research on operating systems. In particular, decentralized operating system (DOS) research [Davies 81] poses a set of problems that make unique demands on an IPC facility. A major factor in this type of research is the lack of specific knowledge of the structure of DOS's. This implies the need for an IPC facility that is flexible enough to permit a wide range of policies to be implemented in the support of DOS experimentation. This need for flexibility, along with the expectation that DOS's will place significant demands on a system's IPC facility, contribute to creating a pair of conflicting requirements for an IPC facility -- i.e., both flexibility and performance. These requirements not only correspond to the expected characteristics of an IPC facility implemented using policy/mechanism separation, but also suggest that other, more common approaches, are less well suited for the job.

4.2.4 Related Work

Although there has been a great deal of work in the general area of IPC [Northcutt 83], relatively little of that work is strongly related to the research outlined in this document. Of the many different types of IPC facilities that exist or have been proposed, few have had flexibility (in the sense of permitting a range of different facilities) as a goal, although some contend that their facility is capable of implementing a wide range of IPC policies [Rao 80].

There have been very few explicit attempts at applying the policy/mechanism separation approach to the design and implementation of operating system facilities. The majority of the work in this area has been applied to other operating system facilities, such as paging, protection, and scheduling [Bernstein 71, Levin 75, Ruschitzka 78]. To the best of our knowledge, there are no instances of IPC facilities explicitly designed and implemented according to the principles of policy/mechanism separation. This is despite the fact that some IPC facilities consist of operations known as "primitives" [Liskov 79]. However, much work has been done in the related, general area of abstraction (e.g., layering) for the design of operating systems and their facilities, including IPC [Reid 80, Zimmermann 80].

4.2.4.1 Design and Implementation of Interprocess Communication Facilities

The IPC taxonomy described as an anticipated output of this research will serve as an illustration of the wide range of the IPC facility design space, while the previously mentioned IPC model will represent the various implementations possible. From these efforts, the range of possible IPC facilities should be apparent, in addition to some of the relationships among existing IPC facilities. There are a number of systems that were designed to permit the implementation of a broad range of IPC policies (e.g., [Fleisch 81] and [Rao 80]).

However, these systems typically provide a complete IPC facility that may have a low level of functionality, but is general enough to permit the implementation of other IPC policies in terms of the fundamental one. Most systems do not make an effort to provide flexibility in their IPC facilities (in the same sense that we have previously discussed), and some of those that do suggest that it be achieved by providing an IPC facility with a low level of functionality [Allchin 82].

There exist a number of systems that purportedly support their IPC facilities with hardware [Cox 81, Ford 77, Giloi 81, Jensen 78b, Jones 79]. The large proportion of these facilities are supported in firmware and not strictly in hardware. While microcode allows IPC facilities to be implemented with one less level of interpretation, the performance benefits are typically not as great as if the support were provided directly by hardware. It should also be noted that, unlike direct hardware support, microcode support for IPC typically maps the structure of some software implementation of the facility directly into microcode. (It is also interesting to note that there also exist more instances of applying hardware support to scheduling than to IPC facilities.)

4.2.4.2 Policy/Mechanism Separation in Operating System Design and Implementation

There exist very few examples that apply the concept of policy/mechanism separation in operating system design and implementation. The first major system which explicitly embodied these ideas, following their inception in the RC4000 multiprocessing nucleus [Brinch Hansen 71], was the Hydra/C.mmp system [Wulf 74]. The Hydra operating system incorporated the principles of policy/mechanism separation, and attempted to make use of these concepts to the largest extent practical in an actual system implementation. However, even in Hydra the principle of policy/mechanism separation was applied to only those facilities that most readily lent themselves to such a treatment, and the intended separation of policy from mechanism in the operating system was (by the designer's own admission) not complete. In subsequent papers on the design and implementation of Hydra [Levin 75], it is acknowledged that some policy was left in the kernel due to the performance constraints imposed by certain portions of the implementation. For example, scheduling policies cannot be carried out entirely outside the kernel, as the cost of a Hydra protection changing procedure call was too expensive to be incurred each time a scheduling decision is to be made. For this reason, the kernel contained parameterized policy programs, causing the separation of policy and mechanism to be incomplete. In addition, Hydra made no attempt to separate policy from mechanism in other cases where the cost in terms of performance was deemed to be too great. Thus in Hydra, the only facilities whose policy and mechanism were (to some degree or other) separated were scheduling, paging, and protection. In retrospect, almost all that was accomplished with regard to policy/mechanism separation was the separation of long range policy from short range policy. Due to the prohibitive cost of crossing protection boundaries to make all of the policy decisions separate of the mechanisms, Hydra "mechanisms" typically carried out short term policy decisions and returned to "policy modules" where longer term decisions were made.

The Intel iAPX 432 implemented what can be thought of as "Hydra on a chip". Because it was in part a VLSI project, the iAPX 432 did not have the same restrictions on its underlying hardware as did the original Hydra project. For this reason, the designers of the iAPX 432 were free to make a different set of design and implementation tradeoffs. However, the iAPX 432 separates policy and mechanism in much the same way and to the same extent as Hydra[7]. This is despite the fact that the same objectives of policy/mechanism separation held for the iAPX 432 designers, and they had the flexibility to provide the hardware support needed to make further policy/mechanism separations practical.

4.2.5 Approach

This section defines the approach that will be taken in this research to achieve the objectives stated in earlier sections. The overall approach is one of "outside-in" development, as opposed to a "bottom-up" or "top-down" methodology. This research will be guided in a top-down fashion by the principles of policy/mechanism separation along with the results of the IPC facility taxonomy and modeling efforts, while the literature survey will provide the raw information for a bottom-up type of effort.

The intent of this research is to explore the effects of applying the principles of policy/mechanism separation to IPC. This will be done by a combination of conceptual and experimental activities. The following is a roughly chronological ordering of the currently identifiable events which will contribute to this research.

4.2.5.1 Survey of the IPC Literature

In order to achieve a solid understanding of the breadth of possible IPC facilities, their implementations, and their system-level implications, a survey of the literature will be performed. This literature survey will include descriptions of existing and proposed IPC facilities, discussions of general operation system issues that relate to IPC facility design and implementation, and papers concerned with interprocessor communication in general. This survey will not be confined to any particular time-frame or subset of the IPC design space. The result of this survey will be an annotated bibliography, which will include a critical analysis of each of the entries. The bibliography described here will serve as the raw material that provides a bottom-up type of impetus to this research.

4.2.5.2 Taxonomy of the IPC Design Space

Based on the data points represented in the bibliography described above, a taxonomical structure of the IPC design space will be created. This will provide a structure for the many example IPC facilities in the literature, provide a means of collapsing these many examples into groups which are isomorphic with respect to their relevant features, and will illustrate the breadth of the IPC design space (in addition to possibly

[7]This might be attributed to the fact that some of the key members of the iAPX 432 project had previously worked on Hydra.

revealing unexplored regions of the IPC design space). It is from this taxonomy that a manageable number of specific examples can be chosen to represent the major types of IPC facilities for use in experiments that attempt to span the breadth of the IPC design space. Furthermore, this process of providing structure to the IPC design space will prove valuable in the later process of defining specific IPC primitives.

4.2.5.3 Conceptual Framework for Representing IPC

In order to compare and to evaluate various dissimilar IPC facilities and their implementations, it is necessary to have some common means of representing IPC in a system context. This calls for the development of a simple model that can easily represent a broad range of IPC facilities. This model must accommodate IPC facilities that provide different functions, are implemented in different ways, and exist at different levels in systems. This tool will aid discussion of the various IPC facilities involved in this research, and will be useful for structuring thought about IPC facilities -- how they are implemented, and how they interact with other operation system facilities.

4.2.5.4 Evaluation Criteria and Methodology for the IPC Primitives

Prior to the specification of a collection of IPC primitives, a means of determining the success (or failure) of the effort must be developed. This will be accomplished by first generating a list of evaluation criteria, and then indicating a methodology for obtaining the necessary information and applying the criteria. The evaluation criteria will largely be derived from the collection of anticipated characteristics of the use of policy/mechanism in IPC facility design. The means by which the criteria are to be applied must also be specified, including the experiments needed to derive a measure of each of the characteristics of interest.

4.2.5.5 Initial Collection of IPC Primitives

At this point, it will be possible to derive a first collection of primitives that will constitute a complete set of IPC mechanisms. The primitives will be synthesized based on experience from the previous tasks, and from the distillation of the many example IPC facilities into a collection of generic IPC activities. The generic IPC activities will be segregated into common groups, and their characteristics will be evaluated to determine if any of the activities could be subsumed as special cases of more general activities. The collection of representative IPC facilities can be viewed as manifestations of a range of IPC policy decisions, and the generic IPC activities are to be transformed into IPC primitives that permit the widest possible range of facilities to be implemented (by the application of different policies). An effort will be made to ensure that the separation of policy from mechanism be as pure as possible (i.e., no policy should be prescribed by the generic activities which are made into primitives). The primitives will, at this point in time, consist solely of the descriptions of their interfaces and behaviors. Thus, the implementation of the primitives should not be a factor in their specification, and implementation artifacts will be strenuously avoided. However, the descriptions of the IPC primitives must be sufficiently detailed to permit correct implementation of them without further guidance.

4.2.5.6 Trial Implementation of the Initial IPC Primitives

Once a full set of IPC primitives has been specified, a series of trial implementations will take place. The purpose of these studies will be to further refine the proposed primitives, and to carry out a wide range of trial policy implementations for the purpose of determining the breadth of coverage of the primitives. These trial implementations will consist primarily of "paper implementations", in order to maximize the number of investigations possible, while minimizing the effort necessary to do so. These experiments will also serve as yet another filtering stage on the set of example facilities (or policies) to be examined.

4.2.5.7 Evaluation and Iteration of the Initial IPC Primitives

Based on the trial implementation experiments, the initial set of primitives will be evaluated according to the defined methodology. As a result of the evaluations, the specifications of the primitives will be modified as needed and the implementation phase will be repeated. This iteration will continue until the primitives are considered acceptable, as judged by the evaluation criteria.

4.2.5.8 Detailed Implementation and Evaluation of the IPC Primitives

The resulting, refined set of IPC primitives will be evaluated in greater depth (although in lesser breadth). These primitives will be implemented in on a local network of personal computers (either Sun's or Perq's) for the purpose of detailed experimentation and analysis. For the most part, the implementation of the primitives will be in a high-level language, and the measurement of the primitives will be limited to that which is necessary to derive the information required by the evaluation methodology. A small number of different policies will be implemented in this series of experiments; the policies chosen will attempt to span the widest range of interesting IPC facilities, with the fewest number of policies. For comparative purposes it may be desirable to implement a policy similar to that of a common IPC facility implemented in some other fashion (e.g., directly implemented, multi-layered, etc.).

4.2.5.9 Investigation of Hardware Support for the IPC Primitives

The specifications of the final set of IPC facilities may be further refined after the detailed implementation studies. In any event, the primitives on which data has been collected in these studies will be used to evaluate the possibility of providing hardware support for them. Various implementation alternatives for the primitives will be investigated, ranging from predominately software to entirely hardware. This work will be carried out primarily as a "paper study", and will make an effort to determine the cost/performance tradeoffs (across the range of practical implementations) for each of the primitives. The evaluation of providing hardware support for the primitives will be based on the measured performance of the detailed implementations of the primitives, the relative impact of the efficiency of each primitive on the performance of the IPC facility created by given policy, and the cost and performance of hardware support for the primitives. The proposed hardware support mechanisms will be specified in a hardware description language.

4.2.6 Contributions

This research will result in *a set of IPC mechanisms* that support the implementation of a wide range of IPC facilities. Another contribution will consist of an *evaluation of the policy/mechanism approach to IPC*, based on implementations of the previously specified mechanisms and a chosen set of IPC policies. An additional contribution will be a determination of the *range of applicability* (and constraints on the use) of a policy/mechanism separation approach to IPC. Further contributions of this research include a *taxonomy* of the IPC design space, and a *logical framework* to represent various implementations of a range of IPC facilities, an evaluation of the degree to which *multiple IPC facilities* can be simultaneously supported, and whether the set of proposed IPC mechanisms can be effectively *supported with hardware* (which would include descriptions of proposed hardware mechanisms).

4.2.6.1 Separation of Policy and Mechanism in Interprocess Communication

The most significant contributions of this work are expected to result from the separation of policy from mechanism in IPC, the creation of a set of IPC primitives, the implementation and evaluation of the primitives, and an evaluation of the viability of policy/mechanism separation in IPC facility design and implementation. In this section we discuss each of these topics.

4.2.6.1.1 A Collection of IPC Primitives

Part of the overall output of this research will consist of the specifications for a collection of IPC primitives. These primitives will be an example of the mechanisms resulting from the application of policy/mechanism separation to the implementation of an IPC facility. This exercise will be particularly valuable as there exist a wide variety of commonly known IPC policies, but no examples of mechanisms for IPC. The primitives will be derived based on an understanding of the range of possible IPC policies, and a determination of a set of mechanisms necessary to implement a wide range of policies. A major effort will be made in defining the functionality of these primitives to maintain a separation of their specification from their implementation. In addition to the specification of each primitive, there will be a justification for each of the primitives.

4.2.6.1.2 Implementation and Evaluation of the Primitives and Policies

Additional contributions will derive from the implementation of the IPC primitives and a selected set of IPC policies. The resulting implementations will be measured, analyzed, and documented. These implementation experiments will be carried out on a local network of Sun Microsystems workstations or Three Rivers Perqs. The evaluation of the primitives, policies, and resulting IPC facilities is to be performed according to a set of criteria established prior to the implementation work. Much of the success of this portion of the overall research effort will be judged by these evaluations. It is planned that the evaluation effort will determine the degree to which the different components exhibit the behavior expected of them, and how the implementations compare to implementations using other approaches.

4.2.6.1.3 Estimation of the Suitability of a Policy/Mechanism Separation Approach

Another expected output from this work is a determination of the overall success of this effort to separate policy from mechanism in IPC. This evaluation is intended to illustrate the conditions under which a policy/mechanism approach is appropriate, the relative cost/benefit tradeoffs of the approach, and the circumstances to which this approach seems best suited. Of particular interest will be the question of how well the separation of policy from mechanism permits an IPC facility to be constructed that meets the needs of a DOS testbed.

4.2.6.2 Applying Structure to the Interprocess Communication Design Space

This portion of the proposed research consists of two main components -- a taxonomically structured representation of the IPC facility design space, and a logical framework to illustrate the manner in which IPC facilities are implemented in systems.

4.2.6.2.1 A Taxonomy of Extant IPC Facilities

This work will generate a taxonomy-like tree structure of characteristics that will present a logically organized representation of the space of existing and proposed IPC facilities (as represented by the open literature). This structure will not be a true taxonomy in the sense of providing both a structure and an interpretation; the primary goal will be a classification to illustrate differences and similarities in the many examples taken from the literature. The example IPC facilities will include many types at all layers, including those in specific systems and those proposed independent of systems. This taxonomy will form a decision tree that will provide a hierarchical organization of the instances of IPC facilities at the leaves. Such a taxonomy will illustrate the range of possible IPC facilities; this will be useful both in determining the coverage of a flexible IPC facility, and in deriving generic IPC facility classes.

4.2.6.2.2 A System Model of IPC

The output from this effort will be a logical framework for structuring thought about IPC in a system context. This framework is loosely referred to here as a model. This model is necessary as there currently does not exist a means of representing the functionality of IPC facilities, and their implementation, in the context of general computer systems. The proposed model is to be used to gain insight into the nature of specific IPC facilities, to help in evaluating the proposed IPC primitives, and to aid in understanding the implications that the IPC primitives have on the other parts of the system. Unlike more formal models, this model will sacrifice rigor in return for a consistent structure that directly represents the concepts of interest. This is as opposed to formal models that are sufficiently expressive to represent the interesting aspects of IPC facilities, but require the information to be heavily encoded (and hence obscured) by the notation.

4.2.6.3 Exploring the Use of Hardware Support for Interprocess Communication

The primary contribution from this work will be an evaluation of the proposed IPC primitives to determine their suitability to being supported in hardware. Each primitive will be examined individually, and an appropriate degree of hardware support will be determined for each one based on cost/benefit assessments (according to a set of environmental assumptions). Where hardware support for primitives is determined to be of the greatest value, hardware support for (or implementations of) primitives will be proposed. The suggested hardware mechanisms will be defined, at least to the register transfer level, by a hardware description language suitable for use in simulation and synthesis efforts.

4.3 References

[Allchin 82] Allchin, James E., Martin S. McKendry, and William C. Thibault.
Interprocess Communication in a Local Area Network Environment.
Submitted for Publication.
1982.

[Barringer 79] Barringer, H., P. C. Capon, and R. Phillips.
The Portable Compiling Systems of MUSS.
Working Paper.
January, 1979.

[Bernstein 71] Bernstein, A. J. and J. C. Sharp.
A Policy-Driven Scheduler for a Time-Sharing System.
Communications of the ACM 14(2):74-78, February, 1971.

[Brinch Hansen 70]
Brinch Hansen, Per.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(4):238-250, April, 1970.

[Brinch Hansen 71]
Brinch Hansen, Per.
RC4000 Software Multiprogramming System.
A/C Regnecentralen, Copenhagen, 1971.

[Cheriton 79] Cheriton, David R., Michael A. Malcolm, Lawrence S. Melen and Gary R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, February, 1979.

[Cox 81] Cox, G., W. Corwin, K. Lai, and F. Pollack.
A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment.
In *Proceedings, Eighth Symposium on Operating Systems Principles*, pages 125-126. 1981.

[Davies 81] Davies, Donald W., Elmar Holler, E. Douglas Jensen, Stephen R. Kimbleton, Butler W. Lampson, Gerard LeLann, Kenneth J. Thurber and Richard W. Watson.
Lecture Notes in Computer Science. Volume 105: *Distributed Systems -- Architecture and Implementation.*
Springer-Verlag, Berlin, 1981.

[Dijkstra 68] Dijkstra, E. W.
The Structure of the T.H.E. Multiprogramming System.
Communications of the ACM 11(5):341-346, May, 1968.

[Fleisch 81] Fleisch, Brett D.
An Architecture for Pup Services on a Distributed Operating System.
SIGOPS-Operating Systems Review 15(1):26-44, January, 1981.

[Ford 77] Ford, W. S. and V. C. Hamacher.
Low Level Architecture Features for Supporting Process Communication.
The Computer Journal 20(2):156-162, May, 1977.
British Computer Society.

[Giloi 81] Giloi, W. K. and P. Behr.
An IPC Protocol and its Hardware Realization for a High-Speed Distributed Multicomputer System.
In *Proceedings, Eighth Annual Symposium on Computer Architecture*, pages 481-493. IEEE and ACM, 1981.

[Habermann 76] Habermann, A. Nico.
Introduction to Operating System Design.
Science Research Associates, Chicago, Illinois, 1976.

[Jensen 78a] Jensen, E. Douglas, George D. Marshall, James A. White, Wallace F. Helmbrecht.
The Impact of Wideband Multiplex Concepts on Microprocessor-Based Avionic System Architectures.
Technical Report AFAL-TR-78-4, Honeywell Systems & Research Center, February, 1978.

[Jensen 78b] Jensen, E. Douglas.
The Honeywell Experimental Distributed Processor-An Overview.
Computer 11(1):137-147, January, 1978.

[Jensen 83] Jensen, E. Douglas.
The ARCHONS Project.
To Appear.
1983.

[Jones 79] Jones, Anita K., Robert J. Chansler Jr., Ivor Durham, Karsten Schwans and Steven R. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 117-127. ACM, December, 1979.

[Kahn 81] Kahn, K. C., W. M. Corwin, T. D. Dennis, H. D. Hooge, D. E. Hubka, and L. A. Hutchins.
iMAX: A Multiprocessing Operating System for an Object-Based Computer.
In *Proceedings, Eighth Symposium on Operating System Principles*, pages 127-136. ACM, December, 1981.

[Lampson 83] Lampson, Butler W.
Hints for Computer System Design.
In *Proceedings, Ninth Symposium on the Principles of Operating Systems.* ACM, 1983.

[Lantz 80] Lantz, Keith A.
RIG, An Architecture for Distributed Systems.
In *Proceedings, Pacific '80.* ACM, November, 1980.

[Levin 75] Levin, R., E. Cohen, W. Corwin, F. Pollack, W. Wulf.
Policy/Mechanism Separation in Hydra.
In *Proceedings, Fifth Symposium on Operating Systems Principles*, pages 132-140. ACM, November, 1975.

[Liskov 79] Liskov, Barbara.
Primitives for Distributed Computing.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 33-42. ACM, December, 1979.

[Liskov 81] Liskov, Barbara and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
Technical Report 210, Massachusetts Institute of Technology, November, 1981.
Computation Structures Group Memo.

[Northcutt 83] Northcutt, J. Duane.
Interprocess Communication -- An Annotated Bibliography.
Unpublished.
1983.

[Parnas 72] Parnas, David L.
On the Criteria to be Used in Decomposing Systems into Modules.
Communication of the ACM 15(12):1053-1058, December, 1972.

[Rao 80] Rao, Ram.
Design and Evaluation of Distributed Communication Primitives.
In *Proceedings, ACM Pacific '80*, pages 14-23. ACM, November, 1980.

[Rashid 80] Rashid, Richard F.
An Inter-Process Communication Facility for UNIX.
Technical Report 124, Department of Computer Science, Carnegie-Mellon University, February, 1980.

[Reid 80] Reid, Lorretta Guarino.
Control and Communication in Programmed Systems.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, September, 1980.

[Ruschitzka 78] Ruschitzka, Manfred.
An Analytical Treatment of Policy Function Schedulers.
Operations Research 26(5):845-863, September, 1978.

[Spier 73] Spier, Michael J.
The Experimental Implementation of a Comprehensive Inter-Module Communication Facility.
In *Proceedings, 1973 Sagamore Computer Conference on Parallel Processing*, pages ?-?. Syracuse University, August, 1973.

[Wulf 74] Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack.
HYDRA: The Kernel of a Multiprocessor Operating System.
Communications of the ACM 17(6):337-345, June, 1974.

[Zimmermann 80]
Zimmermann, Hubert.
OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection.
IEEE Transactions on Communications COM-28(4):425-432, April, 1980.

5. DATE: A Decentralized Algorithm Testing Environment

5.1 Overview

The experimental aspect of our research occurs at both the algorithm and system/subsystem levels. It depends on two complementary components: an interim testbed (see Chapter 7), and a discrete event simulator named DATE, which runs on VAX UNIX.

The DATE system provides a simulation environment where various types of decentralized algorithms can be evaluated. Unlike a large distributed simulation system, DATE was implemented based on a simple set of primitives (or commands). These primitives can support dynamic creation and destruction of processes and interprocess communication primitives.

5.2 Design and Implementation of DATE

5.2.1 Overview of DATE

- The purpose of DATE is to facilitate the experimentation of distributed algorithms in a well instrumented distributed environment.
- DATE provides a set of mechanisms to the user (or experimenter), which can be invoked by primitives (or commands). These mechanisms allow the user to set up the distributed system on which the algorithms are to be tested, and provide tools for experimenting with these algorithms.
- The algorithms being tested can be expressed in one of two ways. The actual code for the algorithms can be written out, or their behavior can be simulated. A combination of the two techniques (part emulation and part simulation) is also possible. The underlying system on which these algorithms execute is simulated.
- The motivation behind the concepts and facilities provided in DATE arises from the need to experiment with distributed algorithms, especially for resource management, in the Archons project. The primitives have been selected after a cursory study of two types of experiments which will be performed on DATE. However, an effort has been made to allow the facility to have wider applicability. Although there has been no attempt to provide a complete set of facilities for a variety of potential users, it is expected that the facility can be extended easily to include other applications.

5.2.2 Functional Specification

5.2.2.1 Overview of Facilities

- DATE provides the ability to concurrently execute multiple user processes on multiple nodes of a simulated global bus network.

- It allows the dynamic creation and destruction of user processes, and communication paths between these processes. This ability is useful in setting up the underlying system on which the algorithm is executed, as well as in implementing the algorithm.

- Processes are defined statically. At the time of the creation of a process, its code must be contained in an executable object file. The code for the process cannot be created during the course of an experiment.

- DATE provides an interprocess message communication facility, which allows three different types of messages. The IPC characteristics of a real distributed system are simulated. Messages encounter unpredictable communication delays. The delay characteristics can be varied by the experimenter. The communication delay for inter-node and intra-node messages will be different in general.

- It provides the ability to set up, start and stop an experiment.

- It provides a recording of all important events of an experiment in an event log file. Information required about a run of an experiment can be recreated from this file. At the conclusion of an experiment, this is the only output provided by DATE.

- It allows the setting of breakpoints in the experiment. These breakpoints can be set at specific points in the code, and also invoked asynchronously from the experimenter's console. At any of these breakpoints, the experimenter can examine the state of the system, alter parameters and system structure, and study the event log file.

- The experimenter can write a postprocessor to extract the required information from the event log file. The postprocessor will be specific to a particular experiment, or to a class of experiments. The number of such routines which will have a wider applicability is unknown. In due course of time, a library of some general postprocessing routines may become available. The postprocessing routines can be executed either at the conclusion of an experiment, or at breakpoints during an experiment.

5.2.2.2 A Scenario for Experimentation

A typical user will follow the steps given below to run an experiment on DATE.

- The code for each type of process that is to be created during the course of the experiment is written out. It is then compiled and linked. At the time of starting DATE, the code for each process exists in a separate object file.

- The postprocessing routines for the experiment are written and compiled.

- DATE is started up. At this time it is in *command* mode, and prompts the experimenter for instructions.

- The experimenter sets up the initial system configuration and the experiment, by creating processes and communication paths to interconnect those processes. This can be achieved directly from the terminal, or by creating a process which sets up the system.

- DATE is allowed to run the experiment for a specified length of time. It is now in *run* mode, and does not respond to the experimenter in this mode.

- The experimenter can asynchronously interrupt the experiment at any time, to bring it back to command mode. This will give the control back to him for any interaction with DATE. Breakpoints can also be set in the process code. Their effect is the same as of the interrupt; i.e., the experimenter can interact with DATE again.

- After interrupting DATE (in either of the two ways described above), the experimenter can use any of the primitives available, modify parameters, change the system structure etc. The postprocessing routines can also be run on the event log file built upto this point in the experiment.

- At the end of the experiment, the DATE system can be terminated. The postprocessor can now work on the log file left by DATE.

- If the system gets wedged in the course of an experiment, the entire system can be killed by sending an appropriate signal.

5.2.2.3 Detailed Specification

- DATE is a message based system. It provides the ability to dynamically create and destroy processes and communication paths. These processes and communication paths can be used to simulate the underlying distributed system being experimented with, and for implementing distributed algorithms.

- The definition of a process is static. The code for all types of processes which will be created during an experiment has to be provided in advance. The definition of a process is known as a *template*. A template gives the type of a process. When a new process is created, its type or template has to be specified.

- Each process is associated with a node. This is to enable the communication system to model the delay characteristics of messages more accurately. On an average, interprocess messages on the same node will have shorter transit times than those across nodes. The nodes are assumed to be connected by buses.

- The IPC mechanism provides three types of message communication.

    - Direct: Single sender, single receiver.

    - Broadcast: Single sender, multiple receivers.

    - Selector: Single sender, any one of a set of possible receivers.

  In effect, the IPC mechanism provides various communication paths for each process. The sender need not necessarily know the type of message being sent.

- *Broadcast* and *selector* types of messages are provided by associating processes with *broadcast sets* and *selector sets*. If a particular message has to be broadcast, it is sent to the appropriate *broadcast set*, and received by all members of that set. Similarly, a selector message is sent to a particular selector set, and received by any one of its members chosen randomly. Membership of a set defines a communication path for a process, on which other processes can send messages to it. Each process can belong to multiple broadcast and selector sets simultaneously.

- *Direct* messages can be sent to any process whose ID is known.

- The IDs of processes and sets provide all the communication paths in the system.

- Each process is associated with a single mailbox on which it receives messages. Mailboxes have fixed sizes which can be set at the time of the creation of the processes. It is also possible to have no limit on the size of a mailbox.

- Messages are prioritized. Priorities define the order in which messages are to be received. Within a priority, the order is first come first serve. A message is *preemptible*, i.e., it can be discarded to make room for a higher priority message, in case the receiver's mailbox overflows. The probability of a message being discarded can be reduced by assigning a higher priority to it. The current system provides 32 levels of priority, the highest being 1 and the lowest 32.

- The underlying communication system simulated by DATE, provides random delays for all messages being sent on the network bus. At present, these delays do not depend on the current system conditions.

- The message passing system does not include any facilities for protection. The system is assumed to be co-operative, not competitive. All the processes are implemented by the same user, and need not be protected against each other. The facility will be used for experimenting with operating system level algorithms. Protection for these algorithms is not essential.

- The following primitives are provided by DATE.

    - **CreateNode**
      This primitive creates a new node, and returns its NodeID.

    - **DestroyNode (NodeID)**
      This primitive removes the specified node from the system, and destroys all processes on that node. It can be used for implementing a processor crash or a node failure of a real system.

    - **CreateProcess (TemplateID, NodeID, MailboxSize)**
      This primitive creates a new process, and returns its ProcessID. The type of process to be created is specified, along with the node on which it is created, and the size of its mailbox. Information regarding various parameters and communication paths to be used by the new process is sent to it by messages.

    - **DestroyProcess (ProcessID)**
      This primitive removes the specified process from the entire system. This includes its removal from its node, and the sets to which it belonged. At this time, some statistics concerning the process being destroyed, are recorded in the log file. These include the

execution of the process, and the number of primitive calls. This information can be used in calculating a better estimate of inter-primitive execution time for the process in a subsequent run of the experiment.

- **CreateSet (SetType)**
  This primitive creates a new set of the specified type (broadcast or selector), and returns its SetID. This primitive is used for creating a new communication path.

- **DestroySet (SetID)**
  This primitive destroys the specified communication path.

- **AddElement (SetID, ProcessID)**
  This primitive is used for adding a process to an existing broadcast or selector set.

- **RemoveElement (SetID, ProcessID)**
  This primitive is used for removing a process from an existing broadcast or selector set.

- **Send (ReceiverID, Priority, Length, MsgContent)**
  This primitive allows a user process to send messages to one or more user processes. All three types of message communication, viz., direct, broadcast, and selector are specified in the same way. The DATE system understands the type of message communication from the ReceiverID. The ReceiverID can either be a SetID or a ProcessID. DATE determines the type of ID and hence the type of message communication specified. The priority of the message, and its length in number of bytes are also specified. MsgContent is a pointer to a buffer in which the message is stored. The send primitive is not a blocking send. The caller does not receive any indication of whether the message reached its destination. He would have to enquire about the message by an end-to-end protocol. In a real distributed system, it is reasonable to assume that the IPC facility is unable to inform the sender about the precise state of his message (if it reached the receiver's mailbox, if the receiver saw the message etc).

- **Receive (Timeout, MsgPointer)**
  This primitive allows a user process to receive messages from its mailbox. The process blocks for at most "timeout" number of seconds, waiting for a message to arrive at its mailbox. The value of timeout can be set to zero, if the user process wishes to poll for a message. The length of the message is returned as the value of the function. If timeout occurred and no messages were received, an error value is returned. If a message is received, its content is placed in the buffer pointed to by MsgPointer. Each user process is associated with a single prioritized mailbox. The receive command returns the first highest priority message to arrive in the mailbox.

- **SetParameter (ParameterName, ParameterValue)**
  This primitive allows the experimenter to define the value of some parameters in DATE, such as the execution time of various primitives, and characteristics of the message communication system. The parameters which can be set in this way are a well defined part of DATE's interface to user processes.

- **Display (ParameterName)**
  This primitive allows an experimenter to see the current value of a system parameter on his console. It is useful in conjunction with the SetParameter primitive.

- **Breakpoint (Length, MsgContent)**
  This primitive is used for setting breakpoints in the code for user processes. Once a break is encountered, the DATE system stops running the simulation, and waits for commands from the experimenter console. This gives the experimenter the opportunity to modify the system (create and destroy user processes, communication paths etc) and system parameters. He can also examine the log file, and run postprocessing routines on it. In its effect, a breakpoint is identical with sending a message to the experimenter's console. The priority of the message is assumed to be the highest (one), and the contents of the message are typed out at the console.

- **Interrupt**
  This primitive is invoked from the experimenter's console by depressing the <DEL> character on the ASCII keyboard. This provides an asynchronous breakpoint. The effect of the interrupt is the same as that of Breakpoint, i.e. of focusing the attention of DATE on the experimenter console. The message sent is the single word "Interrupt".

- **Record (Length, Content)**
  This primitive is analogous to the "Write" statement of a programming language. The contents specified are written out in the event log file.

- **Terminate**
  This primitive is used by the experimenter to terminate the entire DATE system. Statistics related to the execution time and number of primitive calls of all the user processes are entered in the log file.

- **Init (ExecutionTime, NodeID)**
  This primitive must be executed by every user process on startup. It synchronizes the newly created process with the rest of the simulation system. ExecutionTime gives an estimate of the average time taken to execute the instructions between two consecutive primitives in that process. The Init function returns the ID of this new process, and also the ID of its node.

5.2.3 A Sketch of the Implementation

5.2.3.1 The Structure of DATE

In this section, the units comprising DATE are described briefly.

- A central controller process called *Controller* forms the heart of the system. It simulates the concurrent execution of user processes, and provides the interprocess communication mechanism. It also provides the various primitives described in the previous section. It is responsible for running the simulation, recording events, simulating the underlying message communication system, starting the system etc. The main data structures it consists of are the following:

  - A queue of events called the EventQ is provided. In this queue, events are queued for each of the user processes (including the Interface process). These events are to take place in the future, with respect to simulated time. The information given by each entry in the queue is the name of the event, the value of simulated time at which it is to occur, the ID of the process waiting for the completion of the event (if any), and any parameters related to the event (e.g. a pointer to the message in the case of a Send event).

- The controller has tables containing information about the existing user processes and communication paths. Information about a process includes the ID of the node on which it executes, the size of its mailbox, and its status (blocked or active). A list of the members of each set is also maintained.

- Mailboxes are maintained for processes (one per process) from which they can receive messages. A mailbox is a prioritized queue, with an upper limit on the number of entries allowed. Each entry in the queue contains the priority of a message, and a pointer to the message buffer area where the message resides.

- A message buffer area is maintained, in which messages are stored from the time a send event is queued in the event queue upto the time the message is received (or discarded from the system). Besides the message content, the buffer stores the ID of the sender, the receiver and the message.

- An *Experimenter's Interface* process is provided to enable the experimenter at the console to interact with the Controller process. The *Interface* process is very similar to a user process. The controller provides the same primitives to it, as described for the user processes. However, in some ways it acts somewhat differently, e.g., messages sent to or received from it do not take any transit time (in terms of simulated time). The interface process provides a command interpreter. It accepts the commands from the experimenter, and calls the appropriate library subroutines for communicating those commands to the Controller. In the current version of DATE, the command interpreter CI, implemented on the UNIX system at Carnegie-Mellon University, is used. The Interface process also communicates the information received from the controller to the experimenter.

- The library subroutines which are called by the user processes (as well as the interface process) are part of the DATE system. These subroutines provide a user process's interface to the controller process. They hide the details of the UNIX operating system, and the implementation of DATE from the user. The two main tasks performed by these subroutines are:

  - The handling of the UNIX pipes which provide communication between the user process and the central controller.

  - The handling of execution time of user processes. This value of time is used by the central controller in deciding the simulated time at which an event queued by the user process is to occur. In the present version, the library routines keep track of the number of primitives executed, and the total CPU time of the user process, to give an estimate of the inter-primitive execution time of the process.

- An event log file is maintained by DATE which is a dump of all the events taking place.

5.2.3.2 Implementation of Simulation

A discrete event simulation of the distributed environment is performed. The entire system works in "lock-step", such that at any point in real time, only one process is executing. Parallelism is implemented by appropriate handling of simulated time. The central controller allows any one user process to execute at a time. When the executing process makes a call to the DATE system, the request is entered in an event queue,

and the process is suspended. Now, some other process is allowed to execute, and so on. The event queue consists of time ordered events. The process chosen for execution is the requestor of the event at the head of the event queue.

The basic simulation is described below. It consists of taking events from the head of the event queue, executing the primitives, unblocking the processes waiting for those events to complete, getting new events from those processes, and queuing them in the event queue. The various steps taken are described below in some detail, especially with respect to the handling of simulation time.

- Remove the event from the head of the event queue. Call it E1.
- Let the simulated time at which E1 is to occur be T1. Check the current value of the simulated time clock[1]. If the value is less than T1, then update the clock value to T1.
- Record the current value of simulated time, the name of the event, and the ID of the process requesting it, in the event log file.
- Call the procedure F1 which handles E1 type of events, and send it all the parameters associated with E1. Besides executing the required function, the procedure will also find an estimate of the time taken for that function to execute. Let this value of time be T2.[2]
- F1 sends a message to the process P1, which has queued the event E1, and has been waiting since for its completion.
- F1 also records the parameters of the event E1 in the log file, along with the outcome and results of the event.
- Wait for a message from process P1, giving the next event E2 to be queued for it. Process P1 will also send along the value T4 of the time interval between the occurrence of the two events, E1 and E2. The method of finding T4 is explained later in this section.
- Find the value of simulated time T5 at which the event E2 is to occur, by adding T3 = T2 + T4 to T1 (to get simulated time T5).
- Enter E2 in the event queue in priority order according to the value T5.
- Repeat the above operations till the event clock reaches the value set by the experimenter for the termination of the simulation.

The estimate of the time interval between two consecutive events is found by any user process in one of two ways. The two options are known as Default and Override.

---

[1] It has to be either less than T1 or equal to T1.

[2] The time taken to execute each event of type E1 need not necessarily be the same. The procedure can find the time by using a distribution function, which returns different values for the various calls to the procedure. In the present implementation, however, the execution time is a constant, settable by the user.

In the default option, the intention is to determine the time interval by finding the real elapsed time for the CPU to execute the code between the two events (E1 and E2 in the previous example). The default calculation is done by the library subroutines which interface the user process to the rest of the DATE system. This calculation is not visible to the code of the user process. The default mode is based on the assumption that the user process contains the precise code to be experimented with (and not a simulation of its action). In the current implementation, an estimate of the time interval between two events is used, owing to the lack of a high resolution clock for measuring elapsed CPU time on the UNIX system. The library routines keep track of the total execution time of a user process, as well as the number of DATE system calls. These values are recorded in the event log file, when the user process is destroyed. In a subsequent run of the experiment, the experimenter can use the average value of inter-primitive time calculated from this data, to give a good estimate of the elapsed CPU time between two events.

In the override option, the time interval is a simulated value, assigned by an explicit call to a library routine, by the user process. This option is useful when the user process is simulating the behavior of an algorithm or device.

The Interface process works in the override mode, with the value of the time interval between two events always being given as zero. The interface process is treated somewhat differently from other user processes by the DATE system as well. DATE assumes that events take zero execution time when queued by the interface process. As a result of these two facilities, the interface process "hogs" the controller once it gains access to it, because all its events get queued for the current value of simulation time, near the head of the event queue.[3] In effect, in Command mode, each command is executed by using the same simulation mechanism (event queue, event handling procedures, etc), as used in Run mode, but the simulated time does not advance, so that the commands are not a part of the simulation. This occurs at the start and conclusion of experiments, as well as at breakpoints. The interface process relinquishes control by queuing a Receive event with a timeout value of the end of the simulation run.

5.2.4 Present Status

The system has been implemented and been working at the level described in this Section since May 83. The system is currently being ported from VAX to SUN workstations by NOSC.

---

[3] Note that we do not need to worry about other user processes trying to "hog" the system, because we are implementing a co-operating system. The writer of all the user code, and the experimenter at the console will normally be the same person.

6. Decentralized Computer Architecture

6.1 Overview

One of our tenets is that our unconventional decentralized operating system (OS) ought to be reflected in the architecture of the nodes and internode connection facility of the decentralized computer. (Even when the nodes and their interconnection are preordained without consideration of the decentralized OS, knowledge of this OS/hardware interaction enables one to predict the system's suboptimal behavior.)

The first step is to migrate the global resource management from the application portion of each node into a dedicated special purpose machine at each node, designed expressly for executing decentralized global resource management efficiently, yet without taking cycles from the users. Several critical issues must be resolved in order to do this; one is concerned with exploitation of OS and application processing, a task we have begun, as seen in Section 6.2.

Another issue is our controversial position that a complex instruction set processor is not intrinsically without merit, as a few computer designers have recently argued; this debate is discussed in Sections 6.3 and 6.4. Other issues remain untouched this contractual period, but plans are being made for dealing with them as soon as possible.

6.2 Separation of OS and Application Processing

6.2.1 Concurrency Techniques

In this section we discuss in turn each of five classes of operating system functions. At the same time we will provide a number of examples of the ways in which concurrency of operating system and application processing can be exploited, and indicate some of the architectural issues involved.

6.2.1.1 Processes, Synchronization, and Communication

One of the most fundamental and important abstractions supported by almost all modern operating systems is that of a process. Closely associated with processes, and often bound up in their definition, are mechanisms for synchronizing processes and performing interprocess communication. The literature contains abundant examples of proposed and existing systems which provide some form of hardware support for more efficiently implementing the process abstraction (see [Wendorf,J 83] for a survey). A number of these systems exploit concurrency of operating system and application processing to some degree. In particular, it is relatively common for systems to perform the process scheduling task on a separate processor from that on which the application processes are executed.

Process scheduling involves determining, on the basis of priority, waiting time, or whatever, which process, from the set of processes that are ready to run, will next be executed on the Application Subsystem (AS). The computation required to determine which process to run next on the AS can, at least in theory, be performed on the Operating System Subsystem (OSS), and overlapped with execution of the current process on the AS. To do this in practice requires a fairly tight coupling between the OSS and AS. One approach might be to have the OSS and AS share the memory containing the process control blocks, which hold the current status and saved volatile state for the processes that are being executed on the AS. Only the OSS would be permitted to manipulate the queues of process control blocks used to maintain the state of the application processes. Note that this queue manipulation could be done concurrently with AS application processing. Some mechanism would then be needed, such as the exchange jump of the CDC 6600 [Thornton.J 64], which would allow the OSS to force an AS process switch.

Concurrency can also be exploited in supporting fast process switching on the AS, by using a technique which we term *register buffering*. The main impediment to fast process switching is the need to save the volatile state of the current process, and load the state of the next process to be executed. This volatile state includes the general purpose registers, and may also include the virtual memory address map registers.

In the register buffering technique, we duplicate the AS processor's register set, as shown in Figure 6-1. At any given time, the AS only has access to one of the register sets, called the active set. The remaining register set can be freely accessed by the OSS. While the AS is executing the application process associated with the active register set, the OSS can concurrently be saving the other register set in the control block of the previous process and then reloading those registers for the next process to be executed. When at last it is time to switch processes on the AS, the OSS merely causes a switch in the active register set, being sure to synchronize the switch so that it occurs between instructions on the AS. Thus, a process switch will usually occur "instantaneously" and without execution time overhead on the AS.

It should be clear that this register buffering technique can be easily extended to more than two register sets, providing more buffering between the OSS and AS. Once this is done, a further extension to support multiple application processors is quite straight forward. This register buffering technique is significant in several ways:

1. As implied above, if the OSS is usually able to have the alternate register set loaded with the state of the next process to be executed prior to having to switch processes, little or no application processing time will be lost to process switching overhead.

2. If there is usually an alternate, ready to run pro... available, the ability to do a "free" switch to that process will allow even more operating system processing to be overlapped with application processing. A process switch can be done immediately upon every call to the operating system. Processing of the system call can then proceed concurrently with execution of the alternate application process.

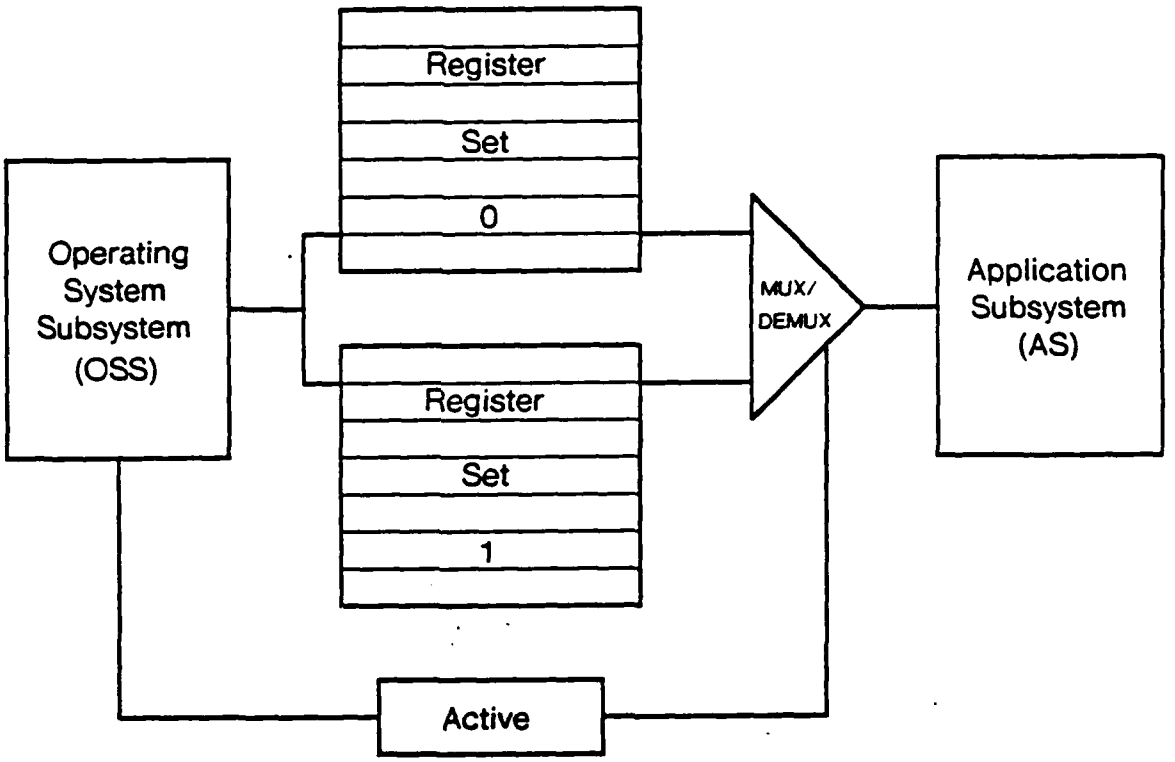


Figure 6-1: An Architecture With Register Buffering

3. Recently there has been considerable attention focused on techniques for using many registers in a processor's architecture, in order to achieve fast procedure calls [Dannenbe.R 79, Ditzel 82, Patterson 81, Radin.G 82, Sites 79]. A recurring problem, however, has been that the increased number of registers, while making procedure calls fast, makes process switching very slow. Register buffering appears to offer an effective solution to the problem of how to provide both fast procedure calls and fast process switching.

The OSS/AS Interface required for the register buffering technique involves the very tight coupling of the two subsystems. The OSS must be able to access and modify the processor registers of the AS. Currently available processors do not permit this type of external access to their "internal" registers. As a result, this technique can only be used in those systems incorporating a custom designed AS, rather than a commercially available application processor. Even the use of bit slice processors to implement the AS requires careful consideration, since many such devices will not permit externally accessible registers to be used.

On the other hand, it should be quite easy to implement the technique for concurrent scheduling of processes, even in systems which use a commercially available processor in the AS. In this case it is only necessary for the OSS to have access to the portion of the AS primary memory which contains the process control blocks and scheduling queues, and have some means of forcing the AS to perform a process switch. Some form of interrupt could be used for this latter purpose.

Shared access by the OSS to the AS primary memory is also the chief requirement for the OSS to be able to support application interprocess communication. The request to send or receive a message could be conveyed from the AS to the OSS via the AS system bus, which is monitored by the OSS. If fast process switching is available, perhaps through the use of register buffering, the OSS can initiate an AS process switch immediately upon receipt of the send or receive request. The OSS can then carry out the request, which primarily involves moving the message from one part of the AS memory to another, concurrently with execution of the new process on the AS. In the absence of fast AS process switching it may be more efficient to simply suspend AS processing until the OSS has handled the send or recieve operation. In such a situation it would be beneficial to have special hardware or microcode in the OSS to make these operations very fast.

The OSS and AS concurrency can also be exploited when creating and destroying processes. Destroying a process can be done very quickly since it is only necessary to mark the process as destroyed. The actual data structure manipulations and other processing required to purge the process from the system can then be carried out by the OSS while the AS continues execution of the process which invoked the destroy function. Since the creation of a process often involves the copying of some state information from the parent to the child, it would be best to switch processes on the AS, assuming fast process switching is available, so that the AS can continue with execution of another process while the create function is being handled. This is similar to the technique used for the interprocess communication functions. Note that the creation and destruction of interprocess communication paths can be handled analogously to creation and destruction of processes. Furthermore, all of these techniques require only that the OSS have access to the primary memory of the AS.

6.2.1.2 Virtual Memory and Protection

The provision of virtual memory and protection in a computer system requires the use of special hardware to perform address translations and protection checks at memory access time. However, the management of pages in memory, and the handling of page faults, is usually left to operating system software. This is another area where concurrency of operating system and application processing can be exploited to provide improved and expanded support for operating system functions. The BCC 500 [Lee.W 74] and SYMBOL [Richards.H 75] are two examples of systems which employed specialized processors to provide extra support for memory management and paging.

As suggested by Ruggiero and Zaky [Ruggiero.M 80], one of the main ways that the OSS could take advantage of the available concurrency is by doing paging out ahead of time. In this way, when a page fault occurs there will be space in memory to accommodate the referenced page without first writing out one of the memory pages. The net saving is one disk access (the write), plus the time required to run the page replacement algorithm, when servicing a given page fault. As a result, a faulting process will become ready to continue execution in approximately half the time as would otherwise be required.

The OSS/AS concurrency also permits the OSS to use more sophisticated page replacement algorithms, without performing the additional computation at the expense of application processing. In particular, the OSS could maintain more complete page fault histories for the AS processes in order to obtain better working set estimations. It may even be possible to anticipate a process' paging behavior and do some amount of paging in ahead of time.

The type of OSS/AS Interface needed to support the virtual memory management techniques outlined above involves shared access to the physical address space of the AS by the OSS. The OSS will need to read pages into that space and write pages from it. The physical address space of the AS can be regarded as a proper subset of the OSS physical address space. One possible arrangement is shown in Figure 6-2.

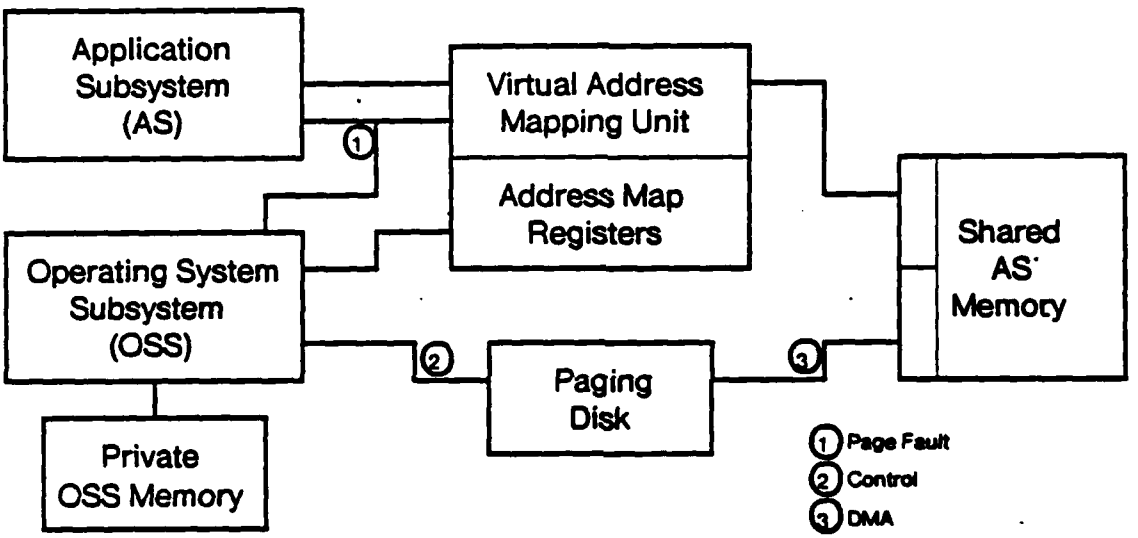


Figure 6-2: An Architecture for Concurrent Virtual Memory Management

In the architecture of Figure 6-2, the OSS controls the virtual address map, which is used to map every memory access made by the AS. Only the OSS can manipulate the Address Map Registers, which specify how the Virtual Address Mapping Unit is to translate the virtual addresses presented to it by the AS into the corresponding physical memory addresses. The OSS also controls the paging disk, which reads and writes pages of the AS physical memory using direct memory access. The AS memory is shown as being dual ported. However, a single port memory on a system bus which permits the paging disk to steal DMA cycles is also feasible. Both the AS and the OSS must be notified of an application page fault. The AS can then unwind the instruction which faulted, and the OSS can then initiate an AS process switch and start handling the page fault.

Note that, in the architecture of Figure 6-2, the OSS has its own private memory which contains the

operating system code, system tables, etc. Also note that it should be possible to use commercially available processors for both the OSS and the AS in such an architecture. It may even be possible to use an existing memory management unit for the virtual address mapping function.

The functions of dynamic memory allocation and deallocation can also benefit from the exploitation of operating system and application concurrency. It is possible for the OSS to do all of the free list manipulations, including concatenation of adjacent free areas, concurrently with the continued execution of the application process. As a result, a more complex free list structure can be used, such as different free lists for the various sizes of blocks, which will permit allocation to be done very quickly. The time required for deallocation will remain very small, almost instantaneous from the application process' point of view, since all that is required is to flag the block as deallocated and then do the actual processing later. These concurrent memory allocation and deallocation techniques require only that the OSS have access to the primary memory of the AS. If the OSS also controls the AS memory mapping unit, it can ensure that a free page is always preallocated, ready for use by the allocation function. This also helps ensure that allocation can be done very quickly.

6.2.1.3 Device Interface

It is now common for an operating system to virtualize the devices it supports by defining a highly abstract interface for each of them. In some systems, such as UNIX [Ritchie.D 74], all devices are made to look like files. In others, such as the IBM System/38 [Hoffman.R 78], all devices are made to look like processes.

By having the OSS provide the device interface, the AS is freed from performing the processing required for low level device handling, such as fielding interrupts and providing buffering. This can be a substantial saving when one considers frequently interrupting devices such as real time clocks. However, it also points out the need for a uniform message addressing mechanism for both application and operating system processes (assuming we are making devices look like processes). An application process on the AS should be able to communicate with a device handler "process" on the OSS in the same manner as it would communicate with any other application process.

The OSS/AS Interface requirements, if the OSS is to provide the device interface, are essentially just those needed for the OSS to support the AS interprocess communication mechanism, as discussed earlier. The only difference here is that sends and receives may involve copying between the shared AS memory and the private OSS memory. However, note that this use of message passing to invoke operating system functions requires that the message communication facility be implemented very efficiently. Otherwise a great deal of AS processing time will be "wasted" in simply calling the operating system.

6.2.1.4 File System

The file system holds a prominent place in most operating systems. As with device interfaces, the file server can be provided as a process to which application processes direct their requests via messages. In this way it is quite straight forward to implement the file server on the OSS, similar to the way device handlers are provided. The same message communication interface as used for device handler invocation can be used for file system requests.

Performing the file system functions on the OSS, concurrently with application processing on the AS, removes a substantial execution overhead from the AS. Furthermore, enhanced capabilities can be added to the file system at no cost to AS processing. Improved file read ahead and write behind can clearly be done concurrently with application processing. Incremental backup of the file system and/or replication of files for reliability is also possible. Disk garbage collection and the ability to run disk diagnostics, without slowing application processing, are two other very significant ways in which the available concurrency can be effectively exploited.

It should be noted that the ability to run diagnostics and tests concurrently with application processing is not unique to the file system. This technique can also be used in the other classes of operating system functions. For example, the virtual memory manager could run a memory diagnostic on each page of the AS memory which it pages out.

6.2.1.5 User Interface

When we speak of the user interface we are referring to the interface provided to the human user when interacting with the system. This could be provided through a process which interacts with the terminal at which the user is located, interprets the input provided by the user, and creates the appropriate processes to carry out the tasks requested by the user.

As with the device interfaces and file system, there are certain advantages to executing the user interface processes on the OSS. Primarily, it permits the interface to be made more sophisticated without reducing the amount of processing power available for executing applications. This is very important now that more and more stress is being placed on the quality of the user interface provided by systems. Many more specialized devices for speech and graphical interaction can be economically accommodated in this way.

6.2.2 Generic Concurrency Techniques

From the work which we have done thus far on developing operating system and application concurrency techniques, as outlined earlier in Section 6.2.1 (Concurrency Techniques), we have noted that our techniques fall into three main classes. These classes represent generic concurrency techniques which can be applied in the implementation of many different operating system functions. The three generic concurrency techniques noted to date are:

1. Precomputation

   The idea here is to anticipate the next occurrence of some function and have most, if not all of the required computation done ahead of time. The register buffering technique is an example. An analogy from another area of computer architecture is the instruction preparation unit used to speed up the interpretation of instructions in many processors.

2. Postcomputation

   Sometimes it is possible to "pretend" to have completed a requested function by simply flagging it as accomplished, and then actually doing the required work afterwards. Concurrent process destruction is an example. This technique is analogous to lazy evaluation.

3. Shifted Tradeoff

   Some functions come in logical pairs, such as dynamic allocation and deallocation of memory. Furthermore there is often a tradeoff between their execution speeds. Depending on the data structure used, one or the other will be fast and its alternate slow. If one of the functions can be handled quickly, due to precomputation or postcomputation, then it pays to shift the tradeoff so that the alternate function is more efficient. For example, memory deallocation can be handled quickly using postcomputation, so we design the free list data structure to permit fast allocation, even though that shifts more computation to the deallocation function.

At present, if we cannot find a way of effectively using one or more of the above techniques when considering the implementation of some operating system function, we must rely on having fast process switching available. If an alternate, runnable application process is available, then that process can be executed while the operating system function is being handled for the current process. Note that this can be regarded as another generic concurrency technique, but in this case it does not improve the execution time of the individual application process on whose behalf the function is being performed. However, it does improve the overall system throughput, since other application processing continues while the operating system function is being executed.

6.3 Instruction Set Architecture Design

6.3.1 Introduction

The increasing size and complexity of processor instruction sets has encompassed additional data types (e.g., floating point, decimal, character strings, arrays, priority queues, linked lists), operating system support (e.g., process management, synchronization, interprocess communication), and compensations for other disabilities (e.g., addressing modes to deal with insufficient instruction address field length). This trend is typified by such popular complex instruction set computers (CISC) as the VAX and the Motorola 68000, and currently culminates in the Intel 432: The intent has been to improve performance, and especially in the case of the 432, to reduce software costs.

Recently an alternative design approach has been widely publicized as the "Reduced Instruction Set Computer (RISC)". Three research machines in particular, the IBM 801 [Radin 83], the Stanford MIPS [Hennessy 82], and the Berkeley RISC I [Patterson 82a], are based on the belief that computers with simpler instruction sets are not only less expensive to design and build, but also offer greater performance than the more traditional complex computers.

We feel that both approaches have merit, but that neither is sufficiently scientific, and we do not find much credible evidence for the claims of either RISC or CISC proponents. Ad hoc designs and implementations have been done but not evaluated, the effects of orthogonal issues have not been separated out, systems which differ in kind have been "compared", important attributes which are difficult to quantify have been presumed not relevant, and complexities have been "removed" by moving them to different places in the system. Unfortunately, many important ramifications of this controversy have remained unappreciated due to unquestioning acceptance of either point of view.

In this paper we briefly discuss some of the topics of contention, note where additional research is needed to attain better understanding, and generally argue for a view of the matter which is broader than just instruction set size and complexity.

We will use the term *RISC* to imply all research efforts concerning reduced instruction set computers. *RISC I* will be specifically used to refer to the research being pursued at Berkeley.

6.3.2 Notions of Simplicity

Perhaps the most fundamental RISC tenet is that the most primitive instructions dominate a computer's activity, and that their performance will be adversely affected by the inclusion of anything more complex in the instruction set. The tenet carries with it an implicit assumption, which is not necessarily true, that any loss in performance is inherently bad. We will discuss this further in Section 6.3.3. This section explores some of the implications and problems that follow from this tenet.

6.3.2.1 Perceiving Distinct Qualities

Unfortunately, the terms *reduced* and *complex* have been contraposed in the context of this first tenet of the RISC philosophy, as Clark has pointed out [Clark 80]. In fact, two orthogonal instruction set dimensions are at issue here: size (reduced vs. massive) and complexity (simple vs. complex). The first dimension concerns the number of instructions (addressing modes, number of possible values in instruction fields in general) that characterize an architecture. The other concerns the functional complexity of the instructions as might be represented by the number of "primitive" operations that would be needed to synthesize them. This dimension is much harder to quantify since mixtures of simple and complex instructions can exist within the same architecture.

It is true that *reduced* and *simple* take on a mutually reinforcing relationship in the context of RISC design, as *massive* and *complex* normally do in the CISC domain. This does not have to be the case. Simplicity means different things to chip designers, computer architects, and all other people involved in the design process. The VAX has often been singled out as being a complex architecture. Yet, from the designer's point of view, the VAX was to be a simple yet massive instruction set. The definition of simplicity used in this context was:

> those attributes (other than price) that make minicomputer systems attractive. These include approachability, understandability, and ease of use [Strecker.W 78].

It is questionable whether or not this goal was achieved, but it will be argued later that, in some ways, the issue of simplicity may not be of prime importance.

6.3.2.2 The Utility of Complex Instructions

RISC proponents warn of detrimental effects due to the use of complex instructions. Nevertheless, the popularity of installing support for specialized functions such as interprocess communication (IPC) seems to be undiminished. The designers of the ELXSI 6400 report [Olson 83], for instance,

> A key architectural feature which allows the operating system to cope with the tremendous variability in its hardware environment is the microcode and hardware implemented message system. The use of messages allowed us to make choices in the CPU and operating system architecture which greatly enhance the effectiveness of additional processors.

But we concur with one of the RISC criticisms of the published accounts of these machines: it is not enough to show that a complex instruction executes faster than an equivalent sequence of primitive instructions. It must also be shown that the net effect is to improve system performance. We believe that this aspect of the problem must be part of the design effort.

Even the premise that primitive instructions always dominate a computer's activities is not universally true. The instruction set interface of machines designed to run operating systems may be solely at the "system call" level, for example.

There are many other computing environments, such as real-time or signal processing systems, where it would be hard to argue against supporting complex functions directly in the computer architecture and implementation. More generally, Radin has written [Radin 83]:

> It is often true that implementing a complex function in random logic will result in its execution being significantly faster than if the function were programmed as a sequence of primitive instructions. Examples are floating point arithmetic and fixed point multiply. We have no objections to this strategy, provided the frequency of use justifies the cost and, more important, provided these complex instructions in no way slow down the primitive instructions [Radin 83].

We subscribe to this statement, but we assert that "*frequency of use*" *is an insufficient criterion* for justifying a given instruction. As Clark and Levy have pointed out [Clark 82]:

> Aggregate statistics alone cannot guide the design of an instruction set intended for different languages and applications. In particular, instructions that are infrequently used overall can be critical for some intended users.

The notion that complex functions slow down the simple actions of a computer seems to be the real problem that prevents us from having the best of all worlds. We believe that serious research efforts in the areas of functional partitioning, instruction interpretation, and distributed decoding will produce computer structures which reduce or eliminate this effect. Until research is directly aimed at this problem, a greater understanding of the scientific truths and principles involved, as opposed to the folklore currently being disseminated, is not possible.

6.3.2.3 *Designing Simple Machines*

The RISC simplicity tenet has a related side-effect in that simpler computers are thought to be easier and faster to design than complex ones. Unfortunately, the comparisons that have been published to substantiate this tenet are based on design times for a student project simple microprocessor versus the design times for some current complex commercial microprocessor products [Patterson 82a]. While these comparisons seem interesting, we do not find them relevant, since the objectives, constraints, and design tasks are significantly different between the academic and industrial environments. Design considerations such as yield, testability, and fault tolerance are not handled in the same way for both contexts. Logistical and administrative factors necessarily imposed by a large organization (e.g., synchronizing simultaneous development of support chips, software development systems, and fabrication facilities) cannot be disregarded. It strikes us as improper to make any such comparisons without first attempting to calibrate the units of measurement.

To make matters more confusing, comparing *the hardware design times of processors* of different scale is misleading since complexity shed by the processor design team could well be encountered by the system software designers or even the applications programmers. The tables of comparisons don't even hint at the tradeoffs. As Justin Rattner has said [Barney 82], "They say that the RISC (I) chip was developed in 6% of the time it took for the 432 ... My response is, 'Yes, and they only did 6% of the job.'" The hardware/software

partitioning of a design begs for a more detailed analysis. In particular, an economic analysis of the hardware/software design cycle tradeoffs would be of strong practical interest.

6.3.2.4 Complexity Migration

One specific form of transferring complexity away from the hardware is by migrating functions to compile time which previously were considered run time activities that were supported by hardware. One of the three criteria for instruction set design in the 801 was that the operation could not be moved to compile time. The 801 approach also utilized a very sophisticated compiler to make some of these tradeoffs (for example, by precomputing functions wherever possible).

This concept of complexity migration is the basis for MIPS [Hennessy 82], which is based on a pipeline implementation having no hardware interlocks. The only means of ensuring proper sequencing of events in this machine's instruction stream is via a *pipeline reorganizer* program. Although a straightforward compiler can be used to generate valid code for this machine, it is only by using the reorganizer that the machine's pipeline can be fully utilized.

Of course, not all complex functions can be moved to compile time. Dynamic program activities, such as garbage collection and bounds checking, must of necessity be done at run time. But as further work is done to evaluate the merits of complexity migration to compile time, computer system designers will be able to make decisions based on evidence rather than educated guesswork.

6.3.3 Importance of the Performance Aspects of Computer Design

Throughout the RISC literature there is a largely unstated but pervasive bias towards those aspects of a computer system dealing with performance. Clearly, if all other attributes are equal, higher performance must be considered an improvement to a machine. However, we believe that it is possible to ascribe too much importance to the performance dimension of a computer system. Since performance is the most quantifiable measure of a machine, it is the most frequently discussed and measured -- not because performance is always inherently so much more valuable than other system parameters, but because benchmarking is the easiest way of comparing system alternatives. It is a mistake to pursue performance blindly without explicitly acknowledging what is being traded for it.

6.3.3.1 What is meant by performance?

System performance can be measured in different ways, and these differences can be significant, because they reflect the fundamental goals of the system. For example, performance can be measured in terms of peak instruction execution rate, a number which may be of interest in deciding the suitability of various supercomputers to some proposed task. Another performance measure is response time, which is of par-

ticular interest in real-time control systems. Yet a third performance measure is average system throughput. We assume that it is this measure which is being discussed in the RISC/CISC literature.

6.3.3.2 Performance vs. Other System Aspects

It is our view that a very wide range of performance is currently available in the marketplace, and that, except for the most demanding applications, a user with sufficient money can buy whatever performance is desired. We agree that the dramatic declines in the price/performance ratio have been largely responsible for the enormous economic growth in the field.

But it strikes us that significant performance gains will be given to us almost free by the semiconductor technologists, first by the constantly improving fabrication process (driving down gate delays), and second by the increasing integration densities that will allow more computational activity to occur without the need to go off-chip. Device technologists can not address other aspects of a computer system, however. For example, even if the military had arbitrarily large amounts of money, they could not buy systems with the level of modularity and expandability they desire, because we architects do not yet know how to provide it at any cost. Conversely, the Japanese Fifth Generation Project requires such large increases in performance that it is commonly assumed that no von Neumann architecture will ever be able to provide it (regardless of the complexity or simplicity of the instruction set). Hence, research is being pursued on multiple processor architectures, where the bulk of the performance results from combining large numbers of processors. Research to deal with these problems should be directed at the interconnect and usage problems as much as the processors themselves.

6.3.4 Ambiguous Performance Claims

Much of the interest generated by the RISC efforts comes from the reported performance improvements. One study [Patterson 82b], for example, lists the execution times of the simulated Berkeley RISC I chip vs. the 68000, the Z8002, the VAX 11/780, the PDP11/70, and the BBN C/70. For every benchmark measured (benchmarks included Ackerman's function, quicksort, and the puzzle program) the simulated RISC I chip promised faster execution times.

We do not find that the performance claims that have been published are conclusive evidence that a breakthrough in the price/performance ratio has been achieved. We will state some of our reservations in the next few sections.

6.3.4.1 All or Nothing

RISC machines have not only a reduced instruction set, but also many other items which affect performance. For example, as we have pointed out earlier [Colwell 83], we believe that the overlapping register window scheme used in RISC I accounts for a substantial amount of the performance expected from that machine, and can be of value to CISC machines as well. Likewise, we would like to know exactly what performance improvement to expect from the compiler and pipeline management techniques used in MIPS. We feel that it would be much more meaningful to compare reduced instruction set machines to CISCs on those aspects which are unique to each, factoring out those which can be utilized in either style.

6.3.4.2 Fair Comparisons

The Intel 432 would seem to be a very promising candidate for close scrutiny in this RISC/CISC controversy. It can be considered an archtypical CISC: it has a complex instruction set (including such instructions as BROADCAST TO PROCESSORS and LOCK OBJECT); it is programmable only in Ada, not assembler; and it is a complete computer system, including an operating system kernel. Although the 432 performance study [Hansen 82a] did not mention RISC I, the same benchmarks were used and it is a simple matter to correlate the two reports to arrive at the conclusion that the 432 runs the benchmarks about two orders of magnitude slower, in general.

But as we pointed out in [Colwell 83], it is important not to overlook other aspects of the 432 that have affected these results. For example, the 432 is an object-oriented machine. This object orientation was provided to support the intended software programming environment, and is an attempt to minimize life cycle cost, not performance per se. However, the other machines measured in [Hansen 82a] were not object oriented, so we are left unsure as to what part of the reported performance loss in the 432 is due to its object orientation. The object facilities cannot be removed from the 432 for comparison purposes, but they can be added in software to other machines to make the results useful.

All benchmarks reported in [Patterson 82b] were coded in the C programming language. We do not object to C as the high-level language (HLL) of choice. Given that machine-dependent aspects of the C language are avoided, this eliminates one source of uncertainty. However, not all comparisons are set up in this way. Our concern is that the 432 has only an Ada compiler. Thus the quality of the Ada compiler, as well as the efficiency of the Ada language itself, are in question here. We feel that these two variables alone render the 432 results inconclusive.

Another interesting aspect of the 432 that may have a negative effect on its single-processor performance is its innate multiprocessing support. This feature was designed at the system level so that processors can be added and automatically utilized without software participation. As far as we know, this is one of the few

system architectures that have ever been produced with this capacity. Intel has estimated that support for multiprocessing takes approximately 13% of the available microcode space on chip, which we take as more evidence that this is not a trivial function to implement. Since the machines to which the 432 is being compared do not have this kind of support, we can only wonder to what extent this feature skews the conclusions that one might attempt to draw.

The 432 also provides some hooks for enhancing system reliability, such as fault handling and functional redundancy microcode. In a naive comparison of instruction execution rates this hidden functionality will appear as unseen baggage, dragging down the machine's performance on the benchmark. Benchmarking is a very interesting exercise, but unless the machines being compared differ only along one major dimension, it is difficult to make a fair comparison. An unfair comparison is inconclusive.

6.3.4.3 Justification and Analysis

The important question is whether these extra features in CISCs such as the 432 contribute enough to have made the apparent performance loss and extra design complexity worthwhile. One study [Cox 83] reports that the 432's support for its interprocess communication primitives do indeed speed up those operations by large amounts over the software approach used in comparable machines. This proves that a SEND can be executed faster if we are willing to devote system resources to it, but it leaves unanswered several other questions of equal importance. How were the complex functions like SEND chosen in the first place? What was gained on a system-wide basis by including these functions? What was the cost, both in resources and in low-level instruction performance? We suggest that one of the tasks facing computer architecture research is to find out how to assign better life-cycle cost models to the systems we build, so that the performance aspects don't receive improper weighting, either positive or negative. Especially in an system architecture as radically different as the 432, it is incumbent upon the designers to carefully justify the design tradeoffs they have made. It seems to us that they have done so at the system level, making the case that object orientation is a goal worth pursuing. However, they do not attempt this same justification at the architecture or implementation levels, nor do they analyze the resulting machine. A good understanding of existing CISCs is not possible without examining the tradeoffs at those levels.

6.3.5 Architectures and Implementations

Since the early sixties, computer designers have found it beneficial to conceptually separate a machine's architecture from its implementation. This dichotomy was useful when trying to decompose the design problem. However, its economic strength came then, as it does today, from software compatibility.

After 20 years of using this concept, a sense of good and bad computer architecture has developed around the notions of purity. To quote from Blaauw and Brooks [Blaauw 82]:

> The architecture must be comprehensible and consistent, so it will be easy to learn and use. The user beholds the whole system. It will be easy for him to master and use it only to the degree that it shows an integrity of concept, a consistency of viewpoint, tying together all the design decisions.

6.3.5.1 The Rules Have Changed

The recent research in RISC concepts has stretched the fabric of some purist computer architecture notions. To begin with, software development costs are still of major concern to most installations, but not at the assembly level. While there are probably more assembly programmers hacking today than we care to know, the world is being dominated by high-level language code. The importance of high-level language programming is reflected by the fact that almost all new general purpose computing machines, RISCs and CISCs alike, are founded on optimizing the execution of compiled code. (This is certainly no new idea in light of the long history of Burroughs high-level language machines.)

6.3.5.2 Departures From Purity

Purist notions notwithstanding, it seems indisputable that blurring aspects of architecture and implementation can often lead to better machines. Again, from the notes of Blaauw and Brooks:

> ... some of the genius of Seymour Cray's work ... lies precisely in his total personal control of architecture, implementation, and realization, and his consequent freedom in making trades across the boundaries.

While the separation of these factors produces conceptually cleaner architectures, and might aid in the partitioning of the design task, RISC research trades these advantages for possible performance improvements. For example, a cache is normally invisible to the software, yet the 801 has explicit instructions for cache control so that the computer does not perform unnecessary cache line loads and stores. Instruction ordering constraints, as imposed by a machine's implementation, are present in the 801, the RISC I, and especially MIPS with its non-interlocked pipeline. These, too, are attempts to optimize a machine's performance by trading across classical boundaries.

Traditionally, microcoding has been a powerful implementation technique for instruction interpretation which has made designing massive/complex machines like the 432 and the VAX tractable. In an interesting twist of concepts, many RISC researchers view their machine architectures as exposing what might otherwise be a hidden vertically coded microengine. While RISC instruction bits drive control lines every cycle via minimal decoding, which is reminiscent of traditional microcoded instructions, such a view ignores the conceptual and cultural differences between macrocode and microcode. Manufacturers have generally not supported machines with such exposed microengines (implementations) since they require individual compilers (and hence, cannot share object code with other machines) and are severely limited in the types of changes that can be made to them after their release. We do not believe that these problems will remain for long, as will soon be explained.

6.3.5.3 Moving to Higher Ground

Since programmers never see the machine-level interface, and since performance gains are possible by mixing architecture and implementation, do "computer families", in the traditional sense, have a place in the futures of computer companies? The designers of computers as diverse as the VAX, the 801, the 432, and RISC I, all wanted their machines to be good targets for compiled code. The natural question to ask becomes: "Why don't computer companies market 'families' of machines that are compatible at the system software level?" The HLL programs would be compatible on "families" of machines that could span a spectrum of price/performance ranges. Each member of the family would be free to trade architecture and implementation features to optimize performance. True, unique compilers would be needed for each family member, since instruction sets would differ, but this issue may decline in importance with the entrance of automated compiler-compilers [Wulf 80].

A possible problem arises with the definition of the common system software interface. Just compare a UNIX manual with almost any machine definition you can find. It seems a hard enough task just to define such a massive interface without having to ensure compatibility of that interface across many machines and their system software. Validation becomes a much larger issue than simply running a suite of test programs. This challenge is not necessarily insurmountable, but it is not well understood at present. The idea is not a new one, just one that has yet to succeed on a grand scale. Probably the largest hurdle will be in overcoming the electro-political status quo that has dictated how computer systems should be structured for the last 20 years.

6.3.6 Conclusion

The RISC advocates have put forth a perspective on processor architecture and implementation which is more coherent and concrete than those which seem to guide most CISC proponents. We feel that this perspective is both interesting and insightful in some ways, yet oversimplified and thus justly controversial.

6.4 Multiple Register Sets

6.4.1 Introduction

The Archons project is an attempt to define and implement decentralized resource management mechanisms at the operating system level and below in a computer. We believe that such a system could benefit by having special processors execute operating system functions. Not only could these processors increase system performance by processing concurrently with their associated applications processors, but they could also be tailored to support the decentralized control functions. We are interested in defining such a machine, which we call Meta.

The characteristics of the Archons operating system are currently being defined. Even without their definition, it is clear that the semantic level of these functions could be quite high. Machine operations to provide direct support for communication, atomic transactions, or resource allocation could be envisioned. This immediately places this machine in the midst of the current heated RISC/CISC (Reduced Instruction Set Computer vs. Complex Instruction Set Computer) controversy [Patterson 80a, Clark 80, Ditzel 80]. In our view, much of the research in this area has been interesting but inconclusive for reasons that we will explain. To help clear our view of this conflict, we are performing two experimental studies that will produce some direct results on particular issues.

For consideration in the Meta machine, we would like to find an existence proof for the performance value of complex instructions in *some* environment. As a means to this end, the first study investigates several aspects of the Intel 432: the extent to which simple instruction performance may be degraded due to complexity; the extent of performance degradation due to object orientation; and the extent to which performance is increased via the machine's hardware/firmware support for complex functions. The experimental method we propose is to migrate the instruction set of the 432, including its object orientation, to more conventional processors. The separation of object-oriented overhead from instruction set complexity issues should make performance evaluation studies of complex processors more relevant and conclusive.

The second study takes a complementary tack on our RISC/CISC concerns. While reduced instruction set advocates have stated reasons why processors with limited functionality might offer improved performance, experimental evidence to support these views is needed to help validate such claims. A few studies have attempted to do this by evaluating particular reduced instruction set architectures. The RISC I architecture [Patterson 82a, Patterson 82c, Foderaro 82] is the subject of one such study. Indeed, the reported performance of this machine is high enough to draw attention. Included in this machine is a mechanism for providing each procedure with its own register set while saving the state of previous procedures in other register sets. This mechanism, called *multiple register sets* here, is used to save the RISC I many memory accesses that it would otherwise have to perform as part of its procedure linkage.

Unfortunately, it is hard to evaluate reduced instruction set concepts based on the results from the RISC I. This is because no attempt is made in these simulations to decouple the performance effects of the reduced instruction set from those of the multiple register sets. Indeed, we believe that instruction sets and multiple register sets have orthogonal effects on performance. If this is so, then multiple register sets could be used to equal advantage in both reduced and complex instruction set architectures. Since we are interested in the experimental support for reduced instruction set concepts, and since we also are curious, as computer engineers, about the effects of multiple register sets on computer architectures, we have started a study to evaluate such effects. It should be noted that the word "architecture" is defined here, as in [Amdahl 64], to

mean the description of "the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

6.4.2 Evaluating Complex Instructions

The design of a computer's instruction set can be driven in many different ways. A machine like the VAX may require an instruction set that maintains some compatibility with previous machines so that the customer base is not alienated. Machines such as RISC I [Patterson 82a] and MIPS [Hennessy 82] attempt to make the best use of the implementation technology, VLSI, hence their architectures reflect concerns such as off-chip delays, amount of design effort, and research goals. The iAPX 432 [Intel 81] design was driven by the desired software methodology: object orientation.

There have even been architectures proposed that defer the instruction set choice until after the computer has been delivered to the user [Brakefield 82, Jensen 77]. This concept is significantly different from that of the common "writeable control store" (WCS). The degree to which a WCS machine can be re-configured is severely limited by the fixed data paths, register sets, and control word conventions characteristic of those machines. To defer the instruction set choice, one must combine the notion of "opcode", which can be viewed as a "hardware procedure call", with the common software procedure call. Programming for such a machine would consist of sequences of function calls, each of which would invoke some hardware or software (or both) in order to effect the desired result. Such a function invocation mechanism is the basis for Meta. However, even for a machine as unconventional as Meta, the issue of support for complex functions arises. Are complex instructions necessary and/or beneficial for such support? The benefits and the costs of including complex instructions are not clearly understood. How, then, do we go about investigating this tradeoff?

We could try looking for precedents. The very large majority of computer instruction sets that have appeared within the last 7 - 10 years are what is CISCs. These are characterized by large numbers of instructions (typically hundreds), a rich set of addressing modes (say, six or more), and the inclusion of specialized instructions, whether for high-level language support (the VAX CASE instruction) or system level support (the 432's SEND operator).

Typically, computer architects strive for high performance, and this has traditionally been the rationale for installing instructions with high semantic content. Very recently, however, a number of articles have appeared [Patterson 80a, Patterson 82a, Hennessy 82] arguing that the way to increased performance lies not in more capable instruction sets but in simpler ones. To support this contention, machines such as RISC I have been designed and some comparisons have been made against complex machines such as the VAX and the 68000.

When we try to assess the RISC arguments we find that some of the perspectives are valuable and persuasive. However, we have reservations about the comparisons that have been published. In particular, we question the effects of operating system overhead, virtual memory, and compiler technology assumed for the benchmarks reported. Another study [Hansen 82b] attempts to demonstrate the overhead associated with a heavily object-oriented architecture such as the iAPX 432. This study found that a 4 MHz 432 runs about an order of magnitude slower than other processors such as the 8 MHz 68000 and the VAX 11/780. But we find it hard to draw conclusions from this for the following reasons:

- The object orientation of the 432 indisputably contributes heavily to the reported performance degradation. But what percentage of the slowdown is attributable to the transparent multiprocessing capability that is built in to the 432? In general, one might expect such a machine to exhibit degraded performance compared to a more conventional uniprocessor, but then the more fair comparison would be between several 432's running in a system vs. other microprocessors.

- Is the performance degradation due to the 432's complex instruction set, or to the object orientation?

- If the simple 432 instructions run more slowly because of complexity (a general RISC argument) do the 432's complex instructions buy any of that performance back? Such instructions were not studied in [Hansen 82b].

The choice of whether or not to make a machine object-oriented must be based on many factors other than performance. We are primarily interested in evaluating the tradeoffs inherent in complex-instruction-set architectures, and are using the 432 as a vehicle to this end. We would, however, like to split the overhead due to object-orientation away from the overhead due to the machine's complexity. We feel that this would be a much more useful evaluation of the tradeoffs made in the 432. We therefore propose the following experiments:

- Advantages of object-oriented support in hardware: According to a basic tenet of RISC philosophy, simple 432 instructions ought to exhibit a performance penalty due to the innate complexity of that machine. We will investigate this by migrating the 432's simple instructions plus their object-oriented overhead to other machines. This will illustrate the effects of complexity on simple instructions without allowing the object-oriented overhead to skew the results. We are also aware of, and have to account for technology, compiler, and data type differences.

- The effects of complex instructions in hardware: According to traditional computer architecture design, migrating software functionality closer to hardware should improve its performance. If that principle holds in the 432, the complex instructions of that machine ought to exhibit improved performance vs. software implementation of the same functions on other machines. We'll investigate this in the same way by moving the 432's complex instructions, including the object-oriented overhead, to other machines.

To date we have developed an ISPS [Barbacci 80] description of the 432 GDP and we are currently adding the complex instruction routines to it. Using the Ada description of the 432 microcode algorithms, we will next begin implementing some of the 432 instructions on the 68000 and the VAX.

6.4.3 Evaluating Multiple Register Sets

Many computer architectures use register sets to provide a fast means of accessing operands using short addresses. Usually, the contents of these registers hold procedure-relevant values. At procedure boundaries, these registers are reloaded for a new set of values. This is done by moving the registers' contents to and from main memory. To reduce the delay of such memory transfers, it is possible to implement several logically identical register sets and to switch among them at procedure boundaries. (This is also true of memory-to-memory machines that hold their procedure-relevant values in areas of main memory.) This type of implementation technique is what we define as multiple register sets (MRSs). It is further possible to reduce memory transfer operations by physically overlapping the logically separate register sets of a calling procedure and its called procedure to allow "free" parameter passing between them. This type of structure will be referred to as an overlapped register set (ORS), and is viewed here as an extended type of MRS.

Ideally, the goal of this second study would be to answer the question:

What are the effects and costs involved
in incorporating multiple register sets
in a computer architecture?

This question could further be broken down into these five issues:

1. In what ways is an architecture's performance changed by incorporating multiple register sets?
2. What changes are necessary to a machine's instruction set and internal structures to support such register sets?
3. How do multiple register sets affect the task of writing a compiler for an architecture?
4. What is the impact of multiple register sets on a machine's need for quick context swaps?
5. How does the choice of high-level language or application affect the usefulness of multiple register sets?

Finding complete answers to all of these questions is beyond the scope of what we wish to accomplish at this time. In limiting the goals of this research, we see the first of these five questions as being most important to address and we plan to give it most of our effort. Although the other areas are of interest, some may not be pursued. In the following sections, each of these five areas of interest will be outlined, with particular detail given to the first.

It should be noted that comparing RISCs and CISCs is not a primary objective of this work. While we hope to learn something about the relative performance and requirements of machines that differ in instruction set complexity, this study concentrates on the relative performance and requirements of the same basic architecture with and without MRSs. Any light shed on the RISC/CISC debate will occur as a secondary result of

this work. This research also does not relate to many other important aspects of the RISC I machine in particular. For example, data path area and man-months of design time are not concerns in this study.

6.4.3.1 Performance Gains

The major reason for incorporating MRSs in a machine is to reduce the number of accesses required of main memory. To do this, each called procedure is given its own set of registers and the most recent procedures' states are kept in the other register sets. In this sense the register sets cache the state of many procedures before the calls (or returns) overflow (or underflow) the register sets' capacity. Many register loads and stores can be saved because a return will often recall a procedure whose state is in a register set, and a call will often find an empty register set available. This is because the procedure call/return patterns of most block structured high-level language programs exhibit an certain amount of "locality." What "locality" means here is that the call depth of a program often varies about some level.

Data caches and stack structures are other approaches used to reduce the number of memory accesses required by a computer. While comparative studies among these approaches and MRSs would be instructive, they are not of primary interest to us. We would ultimately like to evaluate the tradeoffs involved with instruction set complexity. MRSs is a technique, orthogonal to instruction set complexity, that affects the performance of any general-purpose register machine, as does a data cache. This study aims to characterize those performance effects so that they can be removed from the RISC/CISC comparisons of register-oriented machines where they don't belong. Since we are interested in register machine comparisons, stack machines are also not relevant here. Comparisons of stack and register architectures can be found elsewhere [Myers 82].

A technique similar to MRSs is used to reduce process swap time in some machines. In these machines there are also many groups of the architecture's logical register set but each set is used to contain the procedure state of a different process. This way, switching among processes can consist of no more than changing register sets. This is done on machines like the Sigma 7 [Sigma 68], which can have the state of as many as 32 processes in registers, and the Dorado [Lampson 80], which can change process state on every machine cycle.

There are three distinct factors which interact to provide performance gains for MRS machines:

1. fewer memory accesses for storing and restoring procedure state are needed (by having more than one physical register set)

2. fewer memory accesses for passing parameters between procedures are needed (by having ORSs)

3. register sets that are associated with the processor are usually faster than register sets that are stored in main memory (as is done in the BELLMAC-8 and the TI 9900)

These three performance factors are orthogonal in nature. As such, they can be experimentally measured separately. A group of single register set machines can be compared to similar machines that are modified to incorporate MRSs. These comparisons, which would be based on some set of benchmark programs, would gauge the effects of the first factor. This can be done using simulations of the machines and their MRS-modified versions. Either compiler modifications would be needed to create the code for the modified machines, or assembly versions of benchmark programs would be changed by hand to reflect what a compiler could do. These MRS machines can then be further modified so that each register set has a fixed overlap with the register sets of the previous and next procedures. These overlapping registers would be used for passing parameters between procedures, as the RISC I does. Again, simulations can be conducted using the same benchmarks. The results of this set of simulations would give a generalized view of the second factor's impact. Since these simulated machines would only be compared against modified versions of themselves, there is no need to consider differences between machines in register set size or in compiler optimizations or in implementation techniques. Having a uniform method of managing the MRSs is important. Studies of such methods have been made [Tamir 83, Halbert 80]. The results of such studies will be used; no attempt will be made to find independent conclusions in this regard.

The third contributor to the performance of some MRS machines, fast register access, contributes to all machines that have their register sets associated with the processor. Since this research is not concerned with exploring the merits of such register architectures against those of memory-to-memory machines, this factor will not be examined. Other researchers have been interested in this topic [Myers 82] and various machines have been proposed with memory structured in novel ways [Ditzel 82, Patterson 80b].

In the two sets of experiments described above, certain assumptions would have to be made about how the modified versions of the machines are structured. The hazards involved in these decisions cannot be fully anticipated, but their soundness is critical for useful results.

Another approach could be taken to determine the effects MRSs and ORSs on machine performance. It would be possible to run traces of benchmark programs that would tell how many calls were made to each procedure and that would give a call/return profile of the programs. With this information it would be possible to calculate the instruction cycles saved by using MRS and ORS techniques. While this approach would answer our questions, a good simulator with all the necessary event counting mechanisms is available to us, ISPS [Barbacci 80]. With it, simulations can be easily modified to gather any runtime statistic that might be useful.

The choice of benchmarks is an important consideration in this experiment. Benchmarks from the RISC I project could be used with the following advantages: many results already exist, they would provide a means

of checking some of our simulations, and they are written in C, a language with many support tools at CMU. Because they are written in C, these results would also reflect the programming biases supported by this language. (See section 6.2.5 for more on this.) They would also produce results that would be useful to many computer designers. However, we are ultimately interested in the performance of primitives used in the Archons system. Benchmarks that reflect performance on these primitives are under investigation, but may not be developed in time for use in this study.

For these two sets of experiments, we will be using simulations of at least the following machines:

- RISC I: An initial ISPS description of this machine has been created and is being refined. Since it already is an ORS machine, the modification experiments will involve removing its register set overlap and giving it a single register set.

- 68000: An ISPS description for this processor is almost complete. A C compiler exists at CMU that could be modified for these experiments. Unfortunately, the 68000's registers are dichotomized into data and address registers. Care must be given to creating a reasonable MRS version for this reason.

- VAX: An ISPS description for the VAX already exists, as does a C compiler. It is, however, a *very* complex processor and creating a valid modified version might present some problems.

We are also considering using BELLMAC-8 and Nebula [Szewerenko 81] simulations in these experiments.

Two other RISC machines of note are not being considered for this study: the IBM 801 [Radin.G 82] and MIPS [Hennessy 82]. No detailed information on the 801 is available due to its proprietary nature. The MIPS machine presents simulation complexities due to its pipelined nature although it could be a target for later experiments.

6.4.3.2 Machine Support Requirements

The RISC I processor has no special instructions to help it manage its register file. It has an internal trap mechanism that is used to detect underflows and overflows of its register file. It is possible to see this machine as providing minimal support for its on-chip register file, truly in the spirit of RISC. It is also possible to imagine other support mechanisms, in hardware and software, that would contribute to the management of the register file. No experiments are proposed to analyze the possible mechanisms for such support. Instead, it would be useful to see how the machine descriptions used in the previous experiments were modified, or how they might have been, to support MRS machines. In general, this part of the research would consist of categorizing the various means of supporting MRSs and, perhaps, of estimating their impacts on performance and cost.

6.4.3.3 Impact on Compiler Writing

Having MRSs in a machine has only a small effect on compiler writing. Most significantly, the code-generation phase is changed to take advantage of better procedure linkages. Also, a system package might have to be generated that would manage overflow/underflow traps. This code would determine the cause of the trap and would dispatch control to the proper procedure that stores or restores register windows in the case of an internal trap.

When the simulations, which were described in the section on performance gains, are executed, then a working knowledge of the code changes necessary will be developed. Any insight developed into coding differences of significance would be reported in this part of the research. No specific experiments or analyses would be added.

6.4.3.4 Usefulness of Response

With so much more state inside a processor that has multiple register sets, the time required to store and load all of a machine's registers is increased dramatically. The RISC I architecture goes from a minimum of 35 architectural 32-bit registers of state to 125 when its full register file needs to be saved. This increase of internal state brings two questions to mind:

1. Does the increase in process swap time become significant in general multiprogrammed applications or in real time environments?

2. Is there an alternative to having to store the internal state of the processor at each change of context?

The first question can be answered by finding statistics regarding the demands of a variety of systems (rate and distribution in time of context swaps). It should be easy, if these numbers can be found, to find the situations where the increased swap delay is unacceptable. This study will make no efforts to address the second question.

6.4.3.5 Language Effects

Saving memory accesses by having more register state is not always possible. Due to scoping rules, as well as indirect accesses via "pointers" as in C, some variables are not well suited for storage in MRS machines. This leads to either special compiler changes or to special hardware mechanisms that slow such references. The effectiveness of MRSs also depends on the characteristics of the applications to be run on the machine. While such aspects of languages and application determine how well MRSs can be utilized, such generalizations are not goals of this study.

6.5 References

[Amdahl 64] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr.
Architecture of the IBM System/360.
IBM Journal of Research and Development 8(2):87-101, April, 1964.

[Barbacci 80] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek.
The ISPS Computer Description Language
Third edition, Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, 1980.

[Barney 82] Clifford Barney.
Fewer instructions speed up VLSI.
Electronics :101-102, November 17, 1982.

[Blaauw 82] Gerrit A. Blaauw and Frederick P. Brooks Jr.
Computer Architecture.
1982.
Unpublished draft of a book; Fall, 1982.

[Brakefield 82] James Brakefield.
Just what is an Op-Code.
Computer Architecture News 10(4):31-34, June, 1982.

[Clark 80] Douglas W. Clark and William D. Strecker.
Comments on 'The Case for the Reduced Instruction Set Computer,' by Patterson and Ditzel.
Computer Architecture News 6(6):34-38, October, 1980.

[Clark 82] Douglas W. Clark and Henry M. Levy.
Measurement and Analysis of Instruction Use in the Vax-11/780.
In *Proceedings of the Ninth Annual Symposium on Computer Architecture*, pages 9-17. IEEE and ACM, April, 1982.

[Colwell 83] R.P. Colwell, C.Y. Hitchcock III, E.D. Jensen.
Peering Through the RISC/CISC Fog: an Outline of Research.
Computer Architecture News 1(11):44-50, March, 1983.

[Cox 83] G.W. Cox, W.M. Corwin, K.K. Lai, F.J. Pollack.
Interprocess Communication and Processor Dispatching on the Intel 432.
ACM Transactions on Computer Systems 1(1), February, 1983.

[Dannenbe.R 79] Dannenberg, R.B.
An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages.
In *Proc. of 6th Annual Symp. on Computer Architecture*, pages 50-57. IEEE and ACM, April, 1979.

[Ditzel 80] David A. Ditzel and David A. Patterson.
Retrospective on High-Level Language Computer Architecture.
In *Proceedings from the 7th Annual Symposium on Computer Architecture*, pages 97-104. IEEE Computer Society and ACM, May, 1980.

[Ditzel 82] David R. Ditzel and H. R. McLellan.
Register Allocation for Free: The C Machine Stack Cache.
In *Proceedings from the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 48-56. IEEE Computer Society and ACM, March 1-3, 1982.

[Foderaro 82] John K. Foderaro, Korbin S. Van Dyke, and David A. Patterson.
Running RISCs.
VLSI Design III(5):27-32, September/October, 1982.

[Halbert 80] D. Halbert and P. Kessler.
Windows of Overlapping Register Frames.
In *CS292R Final Project Reports*, pages 82-100. University of California, Berkeley, CA, 1980.
unpublished.

[Hansen 82a] P.M. Hansen, M.A. Linton, R.N. Mayo, M. Murphy, D.A. Patterson.
A Performance Evaluation of the Intel iAPX 432.
Computer Architecture News 10(4), June, 1982.

[Hansen 82b] Paul M. Hansen, Mark A. Linton, Robert N. Mayo, Marguerite Murphy, and David A. Patterson.
A Performance Evaluation of the Intel iAPX 432.
Computer Architecture News 10(4):17-27, June, 1982.

[Hennessy 82] John Hennessy, Norman Jouppi, John Gill, Forest Baskett, Alex Strong, Thomas Gross, Chris Rowen, and Judson Leonard.
The MIPS Machine.
In *Proceedings of the Spring CompCon*, pages 2-7. IEEE, February, 1982.

[Hoffman.R 78] Hoffman, R.L. and Soltis, F.G.
Hardware Organization of the System/38.
In *IBM System/38: Technical Developments*, pages 19-21. IBM GS80-0237, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 544-546.

[Intel 81] Intel Corporation.
Introduction to the iAPX 432 Architecture
3065 Bowers Ave., Santa Clara, Calif. 95051, 1981.
Manual 171821-001.

[Jensen 77] E. Douglas Jensen and Richard Y. Kain.
The Honeywell Modular Microprogram Machine: M^3.
In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 17-28. March 23-25, 1977.

[Lampson 80] Butler W. Lampson and Kenneth A. Pier.
A Processor for a High-Performance Personal Computer.
In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 146-160. IEEE/ACM, May 6-8, 1980.

[Lee.W 74] Lee, W.K.
The Memory Management Function in a Multiprocessor Computer System - A Description of the BCC 500 Memory Manager.
Technical Report R-2, The Aloha System, Task II, Dept. of Electrical Engineering, Univ. of Hawaii, September, 1974.

[Myers 82] Glenford J. Myers.
Advances in Computer Architecture.
John Wiley and Sons, 1982.
2nd Edition.

[Olson 83] Robert A. Olson, B. Kumar, Leonard E. Shar.
Messages and Multiprocessing in the ELXSI System 6400.
In *Proceedings of the Spring 1983 CompCon.* IEEE, March, 1983.

[Patterson 80a] David A. Patterson and David R. Ditzel.
The Case for the Reduced Instruction Set Computer.
Computer Architecture News 8(6):25-33, October, 1980.

[Patterson 80b] David A. Patterson and Carlo H. Sequin.
Design Considerations for Single-Chip Computers of the Future.
IEEE Transactions on Computers C-29(2):108-116, February, 1980.

[Patterson 81] David A. Patterson and Carlo H. Sequin.
RISC I: A Reduced Instruction Set VLSI Computer.
In *Proceedings from the 8th Annual Symposium on Computer Architecture*, pages 443-457. IEEE Computer Society and ACM, May, 1981.

[Patterson 82a] David A. Patterson and Carlo H. Sequin.
A VLSI RISC.
Computer 15(9):8-21, September, 1982.

[Patterson 82b] D. Patterson, R.S. Piepho.
RISC Assessment: A High Level Language Experiment.
In *9th Annual Symposium on Computer Architectures.* April, 1982.

[Patterson 82c] David A. Patterson and Richard S. Piepho.
Assessing RISCs in High-Level Language Support.
IEEE Micro 2(4):9-19, November, 1982.

[Radin 83] George Radin.
The 801 Minicomputer.
IBM Journal of Research and Development 27(3):237-246, May, 1983.

[Radin.G 82] Radin, G.
The 801 Minicomputer.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 39-47. ACM, March, 1982.

[Richards.H 75] Richards, H., Jr. and Oldehoeft, A.E.
Hardware-Software Interactions in SYMBOL-2R's Operating System.
In *Proc. of 2nd Annual Symp. on Computer Architecture*, pages 113-118. IEEE and ACM, January, 1975.

[Ritchie.D 74] Ritchie, D.M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[Ruggiero.M 80] Ruggiero, M.D. and Zaky, S.G.
A Microprocessor-Based Virtual Memory System.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 228-235. IEEE and ACM, May, 1980.

[Sigma 68] *SDS Sigma 7 Computer Reference Manual*
Scientific Data Systems, 1968.

[Sites 79] Richard L. Sites.
How to Use 1000 Registers.
In *Proceedings of the CalTech Conference on VLSI*, pages 527-532. January, 1979.

[Strecker.W 78] Strecker, W.D.
VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family.
In *Proc. of National Computer Conf.*, pages 967-980. AFIPS, June, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 716-729.

[Szewerenko 81] Leland Szewerenko, William B. Dietz, and Frank E. Ward, Jr.
Nebula: A New Architecture and Its Relationship to Computer Hardware.
Computer 14(2):35-41, February, 1981.

[Tamir 83] Yuval Tamir and Carlo H. Sequin.
Strategies for Managing the Register File in RISC.
To be published in the IEEE Transactions on Computers, 1983.

[Thornton.J 64] Thornton, J.E.
Parallel Operation in the Control Data 6600.
In *Proc. of Fall Joint Computer Conf., Pt. 2*, pages 33-40. AFIPS, 1964.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 730-736.

[Wendorf.J 83] Wendorf, J.W.
Hardware Support for Operating System Architectures.
1983.
In preparation, Computer Science Department, Carnegie-Mellon University.

[Wulf 80] Wm. A. Wulf.
PQCC: A Machine-Relative Compiler Technology.
Technical Report CMU-CS-80-144, Carnegie-Mellon University, 1980.

7. Interim Decentralized System Testbed

7.1 Overview

The purpose of the Archons interim testbed is to support the implementation and experimental evaluation of algorithms for decentralized resource management and to support the development of a prototype decentralized operating system, ArchOS, incorporating an integrated set of these algorithms.

The long-range plan requires that Archons project hardware be specifically designed to provide support for the ArchOS software (see Chapter 6). However, at this point we do not have a version of the ArchOS software on which to base any hardware support requirements. As a result, we are constructing an interim testbed facility on which experiments (mostly dealing with software, but not excluding hardware) can be performed. This system has been designed to supply some general capabilities in order to support the development of the initial ArchOS operating system without requiring that we design special-purpose hardware.

7.2 System Selection

During the period from January to May 1983, we evaluated alternatives for our testbed system. Since ArchOS is a decentralized operating system, it was decided that the interim testbed hardware should be a collection of processing nodes interconnected by an Ethernet to form a local area network. Based on various hardware and software considerations, including availability of compatible off-the-shelf hardware and software, and compatibility with other research efforts, we chose the Sun Microsystems, Inc. (SMI) Workstation as the processing node for the system.

The Suns fulfill the general requirements that were formulated at the beginning of the interim testbed effort, specifically:

- Motorola 68000 processor;
- UNIX operating system;
- high-level language support;
- 10Mbit Ethernet;
- hardware expandability.

The Sun workstation uses the Motorola MC68010, a version of the 68000 with support for virtual memory management. The 68000 is a popular processor with a large software base.

Suns are supplied with the DARPA standard Berkeley UNIX, version 4.2bsd, including system source code. This system provides a powerful software development environment. In addition, it has extensive networking facilities, which are useful for supporting distributed software experiments. The 4.2bsd system also runs on the DEC VAX-11 computers in the Computer Science Department, so we can take advantage of locally developed software, and of project members' experience with the system.

Compilers for C, Berkeley Pascal, and Fortran 77 are supplied with the Berkeley UNIX 4.2bsd system. Both Berkeley Pascal and Fortran 77 can call C routines, and the runtime support systems for these languages are written in C, so modifying the operating system dependent parts of the runtime support to use BBN's CMOS system calls will make it possible to write programs to run under CMOS in any of these languages. CMOS is simple operating system kernel written in C that provides low-level support for multiple processes, interprocess communication/coordination, asynchronous I/O, memory allocation, and system clock management.

The 10Mbit Ethernet is a standard high-speed inter-node communications medium. The TCP/IP Ethernet software supplied with 4.2bsd will allow Ethernet file transfers between the testbed system and the Computer Science Department's machines. It also supports network virtual disks, making it possible for a single disk server to support a number of diskless workstations.

Since the Multibus is used as the system bus, the Sun workstation is expandable. We may easily acquire off-the-shelf hardware or build new boards which can be added to the system; also, since Multibus supports multiple bus masters, we have the capability to add a second CPU card in a single workstation, thereby more closely approximating the hardware of the final Archons testbed facility.

The second major reason for selecting the SMI hardware was to facilitate the sharing of software with other experimenters using similar development systems. In particular, we are interested in cooperating with the work being carried out by Bolt, Beranek and Newman in the area of distributed operating systems. We are beginning to examine their C70/UNIX Distributed Operating System to determine how it may be moved into the Archons interim testbed system environment. The first step will be to examine BBN UNIX and compare it to Berkeley UNIX 4.2bsd.

We anticipate that some experiments will not require and might be hindered by the presence of a large and complex operating system. For this reason, we intend to provide an intermediate level of support between the bare machine and the full 4.2bsd system. For this purpose, we currently plan to use the BBN CMOS system. CMOS is an open operating system kernel, in the sense that there are no security barriers between the OS and the user program. This feature gives us full flexibility for low-level software, while providing a minimum level of system services to programs that need them.

We also recognize a need for performance measurement tools in both 4.2bsd and CMOS. Aside from a simple execution profiler included in UNIX, we plan to investigate systems that are better suited for distributed performance monitoring or debugging. One possibility that seems promising is a distributed monitoring system developed at CMU for the Cm* project [Snodgrass 82].

7.3 Current Status and Future Plan

We are beginning the integration work required to construct the Archons interim testbed facility. This work is being carried out on three Sun Workstations that have been loaned to us until the hardware to be purchased specifically for our work arrives. We have connected the interim testbed system with the CMU CSD's ethernet cable so that the testbed system can interact with the rest of the CMU CSD computing facilities. The most important on going tasks are: to learn how new hardware can be added to the system (in particular, to learn how to write device drivers for UNIX 4.2bsd); and to bring up a small, stand-alone operating system kernel to be used for low-level operating system experimentation.

One unresolved issue is how changes in the execution environment can be handled most efficiently. Although it is certainly possible to reboot the hardware with a different environment (such as the UNIX operating system, the C70/UNIX DOS, or a ArchOS standalone experimental environment) each time a change is desired, we hope to avoid such an inconvenient approach. But, if we can't avoid this approach, then we must make the method as convenient as we can. For instance, it may be feasible for some processing nodes to be used for program development while others are running experiments.

As we move on to our search for appropriate hardware architectures for a decentralized computer system, it may be possible to use our interim testbed to simulate alternative hardware configurations. For example, routines can be written that will make it appear that the nodes are connected by several buses, allowing experimentation with ArchOS handling of individual "bus failures and recovery". We could also insert a context swapping mechanism that would give the appearance of several processors at each node to allow us to test that ArchOS can actually tolerate OS concurrency, even at an individual node. Our initial ArchOS will actually execute directly on the interim testbed hardware, but our work on experiment control and monitoring tools will be done with the objective of being transportable to a simulated hardware situation.

7.4 References

[Snodgrass 82] Richard Snodgrass.
Monitoring Distributed Systems: A Relational Approach.
Technical Report CMU-CS-82-154, Carnegie-Mellon University, 1982.

Appendix

A. Annotated Bibliographies A-1

A.1. Decentralized Operating Systems A-1
A.2. Interprocess Communication A-22
A.3. Hardware Support for Operating Systems Architecture A-66

B. Additional Work on Transactions B-1

B.1. Synchronizing Shared Abstract Types B-1
B.2. Transactions: A Construct for Reliable Distributed Computing B-35

A. Annotated Bibliographies

A.1. Decentralized Operating Systems

[Aiso 75] Aiso, H.; Tokuda, H.; Ishizuka, A.; Kamibayashi, N.; Takeyama, A.
The System Software for KOCOS.
In *Proceedings of the IFIP TC-2 Working Conference on Software for Minicomputers*, IFIP, September, 1975.

Abstract

This is a study on the system software of the minicomputer complex, KOCOS (Keio-Oki's Complex System). The purpose of this system is to first realize resource and load sharing in the heterogeneous minicomputer complex. Finally, the purpose is to realize parallel processing through organic integration of resources. This system is characterized by the following two points: first, the system software is composed of two modules. One, called System Scheduler, controls all the static system resources in a centralized manner, and the other, called Local Operating System, distributively takes care of the execution of processes on each minicomputer. Secondly, the interprocess communication facility has been realized through positive utilization of microprocessors and is rich both in flexibility and expandability. This paper outlines the system configuration, structure of System Scheduler and Local Operating System, and the interprocess communication facility.

[Allchin 83] James E. Allchin and Martin S. McKendry.
Support for Objects and Actions in Clouds.
Technical Report GIT-ICS-83/11, Georgia Institute of Technology, May, 1983.

Abstract

This status report describes the current work of the Clouds project at Georgia Tech. The Clouds project is studying techniques for construction of reliable computing systems in environments of distributed machines interconnected by local area networks. This report emphasises the functional requirements for architectural support. To support reliability, the architecture supports *objects* and *actions*. Objects are instances of abstract data types. They provide a basis for building system components and for controlling the behavior of a system when failures occur. Atomic actions are a means of dynamically grouping invocations of operations on objects into units of work that either complete in their entirety or do not have any effect whatsoever. Recovery mechanisms assist in maintaining this abstraction and synchronization mechanisms control interactions between actions.

[Almes 83] Almes, G. T.
Integration and Distribution in the Eden System.
In *IEEE International Workshop on Computer Systems Organization (New Orleans LA)*, pages 62-71. IEEE, March 29-31, 1983.

Abstract

Although locally distributed computer systems are becoming increasingly common and attractive, operating systems designers have paid little attention to the special needs and opportunities of these systems. The Eden project is one of the few attempts to design an operating system appropriate to these needs and opportunities. This paper describes the approach taken by the Eden project in designing a system both appropriate to a specific class of computer systems and supportive of a modern software distributed hardware base and a logically integrated operating system.

[Andre 82] Andre, J. P.; Petit, J. C.; Derriennic-Le Corre, H.
Dynamic Software Reconfiguration in a Distributed System (Galaxie).
In *IEEE International Conference on Communications. ICC '82: The Digital Revolution (Philadelphia PA)*, pages 5G.4.1-5G.4.5. IEEE, June 13-17, 1982.

Abstract

Distributed architectures are becoming very attractive in building complex softrware systems, such as control for switching systems. To obtain the benefit of the inherent advantages of such architectures, e.g. graceful degradation, extensibility and adaptability, basic concepts of distribution in operating systems must be specified and experiments performed. This paper deals with a system (Galaxie) aiming at an experimental implementaion of these concepts, mainly in the fields of dynamic software allocation. Moreover, in order to provide levels of abstraction with regard to the organization of the underlying hardware and network architecture, the authors present a modular and hierarchical operating system model.

[Applewhite 82] Applewhite, Hugh L.; Garg, Roli; Jensen, E. Douglas; Northcutt, J. Duane; Sha, Lui. *Decentralized Resource Management in Distributed Computer Systems.* Technical Report RADC-TR-81-203, Rome Air Development Center, Griffiss AFB, NY, February, 1982.

Abstract

This is the first technical report from the Archons project, which is performing research in the science and engineering of 'distributed computers'. By this we mean a computer having a highly decentralized (e.g., consensus) resource management at every level of abstraction from the executive down. This report provides a snapshot of several incomplete, ongoing investigations: decentralized synchronization; the requirements for simulation of decentralized resource management algorithms; and the facilities to be provided by a decentralized executive. We begin with a summary of our views on decentralized resource management and control, and the implications of physical communications on control (especially at the executive level). Then we briefly survey several other distributed system projects. This brings the Archons project into closer focus, as their orientations and objectives are considerably different from ours. Synchronization (the induction of a common, consistent ordering on events) is the essence of decentralized control. New concepts and techniques are required to achieve synchronization in distributed computers without reliance on any centralized entity such a semaphore, monitor, sequencer, or bus arbiter.

[Ayache 82a] Ayache, J. M.; Courtiat, J. P.; Diaz, M.; Michelena, J. Software and Protocols in REBUS. A Distributed Real-Time Control System. In *Software for Computer Control 1982, Proceedings of the Third IFAC/IFIP Symposium (Madrid, Spain)*, pages 147-153. IFAC/IFIP, October 5-8, 1982.

Abstract

REBUS is a robust and fault tolerant cooperation system for a local real time control microcomputer network. It is being developed at the LAAS in connection with the industrial real time control system MODUMET 800 of Schlumberger-Europe. Based on a general hardware architecture, the design of REBUS emphasizes the aspects of cooperation and fault tolerance as required in local real time control networks and it is primarily concerned with the problems of specification, validation, and implementation of some standard and specific protocols. After a short presentation of the hardware architecture, the various software levels are described; they include the operating system kernel of the processors, the line, network and transport layers, and the remote call mechanism. Finally, a tool, the observer developed for protocol debugging and measure purposes is also presented.

[Ayache 82b] Ayache, F. M.; Courtiat, J. P.; Diaz, M. REBUS, A Fault-Tolerant Distributed System for Industrial Real-Time Control. *IEEE Transactions on Computing* C-31(7):637-647, July, 1982.

Abstract

Presents a fault-tolerant distributed system designed for real-time control applications (REBUS), which is one of the research basis of the industrial real-time system MODUMAT 800. It is made up of functional units, i.e. programmable multiloop regulators and operator displays, linked together by a communication structure. The communication hardware consists of a set of serial bus interface boards, one per functional unit, loosely coupled together by a double serial bus and linked to their functional units by a private parallel bus. The communication software, implemented on each interface board, provides a distributed executive based on a reliable link protocol and a robust bus allocation mechanism. Different fault-tolerant mechanisms are implemented in order to achieve the dependability requirements of industrial control systems.

[Ball 76] Ball, J. E.; Feldman, J.; Low, J. R.; Rashid, R.; Rovner, P. RIG, Rochester's Intelligent Gateway: System Overview. *IEEE Transactions on Software Engineering* SE-2(4):321-328, December, 1976.

Abstract

Rochester's Intelligent Gateway (RIG) system provides convenient access to a wide range of computing facilities. The system includes five large minicomputers in a very fast internal network, disk and tape storage, a printer/plotter and a number of display terminals. These are connected to larger campus machines (IBM 360/65 and DEC KL10) and to the ARPANET. The operating system and other software support for such a system present some interesting design problems. This paper contains a high-level technical discussion of the software designs, many of which will be treated in more detail in subsequent reports.

[Ball 82] Ball, J. E.; Barbacci, M. R.; Fahlman, S. E.; Harbison, S. P.; Hibbard, P. G.; Rashid, R. F.; Robertson, G. G.; Steele, G. L. Jr.
The Spice Project.
Technical Report, Computer Science Research Review, Carnegie-Mellon University, 1982.

Abstract
The long-range aim of the Spice project is to create a departmental personal computing environment that will be usable through the 1990's. Development and hardware aquisition will be spread over about four years, so that by 1985 most departmental research and ordinary computing will be performed on personal computers. The computers will be connected together by one or more high bandwidth local networks, to access each other and to access central services such as printers and file servers. Gateways will be available to other networks, such as the ARPAnet. Specialized devices will be attached to the network for particular projects. Also available will be the present timesharing facilities, which will survive at least though a transition period.

[Bane 81] Bane, R.; Stanfill, C.; Weiser, M.
Operating System Strategy on ZMOB.
In *1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management (Hot Springs VA)*, pages 125-132. IEEE, November 11-13, 1981.

Abstract
The ZMOB multiprocessor computer will use a distributed operating system with a host controller. The operating system, called MOBIX, gives to the user the image of using an ordinary UNIX system but with truly parallel process execution. Individual ZMOB processors can communicate directly with each other, but hard system calls and references to global names are referred to the host for action. The interprocess communication protocols are sufficiently general to allow many kinds of programs, including both synchronous and asynchronous applications.

[Baskett 77] Baskett, F.; Howard, J. H.; Montague, J. T.
Task Communication in DEMOS.
In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23-31. ACM, November, 1977.

Abstract
This paper describes the fundamentals and some of the details of task communication in DEMOS, the operating system for the CRAY-1 computer being developed at the Los Alamos Scientific Laboratory. The communication mechanism is a message system with several novel features. Messages are sent from one task to another over *links*. Links are the primary protected objects in the system; they provide both message paths and optional data sharing between tasks. They can be used to represent other objects with capability-like access controls. Links point to the tasks that created them. A task that creates a link determines its contents and possibly restricts its use. A link may be passed from one task to another along with a message sent over some other link subject to the restrictions imposed by the creator of the link being passed. The link based message and data sharing system is an attractive alternative to the semaphore or monitor type of shared variable based operating system on machines with only very simple memory protection mechanisms or on machines connected together in a network.

[Berg 82] Berg, H. K.; Smith, M. G.
A Distributed System Experimentation Facility.
In *Proceedings of the 3rd International Conference on Distributed Computing Systems (Miami/Fort Lauderdale, FL)*, pages 324-329. IEEE, October 18-22, 1982.

Abstract
Describes the Distributed System Testbed (DTS) developed at the Honeywell Corporate Computer Sciences Center. The motivations for the use of experimentation facilities in distributed processing research are recalled, and design of DST are summarized. The concepts which are realized by DST are summarized. The concepts which are realized by DST are discussed with emphasis on the instrumentation facilities and experiment control. Both the system hardware and the system software are described. The discussion of the system hardware highlights the node hardware, the interconnection hardware and the experiment timing affordable by these components. The discussion of the system software concentrates on the structure and concepts of the operating system kernel and the applicability of the kernel primitives to experimentation with and instrumentation of the testbed.

[Bernstein 79] P. A. Bernstein, D.W. Shipman and J. V. Rothnie, Jr.
Concurrency Control in SDD-1: A System for Distributed Databases: Part I: Description and Part II: Analysis of Correctness.
Technical Report CCA-03-79 and CCA-04-79, Computer Corporation of America Technical Reports, January, 1979.

[Bevan 80] Bevan, S. J.
A Preliminary Implementation of POSER.
Technical Report DRIC-BR-76603, Defence Research Information Centre, Orpington, England, September, 1980.

Abstract

POSER is a process organisation to simplify error recovery intended for use in fault tolerant, distributed computer systems running real-time programs. This memorandum describes the process organisation used in POSER and how the organisation has been experimentally implemented in a multi-computer simulation. Application program design has been studied by producing a large radar tracking program which runs on the POSER simulation. A version of the radar program exists in MASCOT and some comparisons of the two complete programs have been make. Finally, some broad comparisons of the MASCOT and POSER methods are made.

[Birrell 82] Birrell, A. D.; Levin, R.; Needham, R. M.; Schroeder, M. D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-274, April, 1982.

Abstract

GRAPEVINE is a multicomputer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines, and services; for authenticating people and machines; and for locating services on the internet. This paper has two goals: to describe the system itself and to serve as a case study of a real application of distributed computing. Part I describes the set of services provided by GRAPEVINE and how its data and function are divided among computers on the internet. Part II presents in more detail selected aspects of GRAPEVINE that illustrate novel facilities or implementation techniques, or that provide insight into the structure of a distributed system. Part III summarizes the current state of the system and the lessons learned from it so far.

[Blair 82] Blair, G. S.; Hutchison, D.; Shepherd, W. D.
MIMAS-A Network Operating System for Strathnet.
In *Proceedings of the 3rd International Conference on Distributed Computing Systems (Miami/Fort Lauderdale, FL)*, pages 212-217. IEEE, October 18-22, 1982.

Abstract

Recent technological advances and developments in user requirements have led to the recognition of a new branch of computer science, that of distributed systems. A great deal of research is required before their potential benefits can be fully realised. At Strathclyde University, research into distributed systems has followed a bottom-up layered approach. The first stage was the design of an Ethernet-like local area network called STRATHNET. This was followed by the development of an interprocess communication service employing the notion of a port which provides a testbed for experimentation into distributed operating systems design. The distributed operating system will primarily integrate a number of departmental PDP-11's running the UNIX operating system and will reside in a series of layers above the UNIX kernel. The main design criteria for the system are ease of incremental growth, high availability and reliability. This paper outlines the design of the MIMAS network operating system.

[Boebert 78a] Boebert, W. E.; Franta, W. R.; Jensen, E. D.; Kain, R. Y.
Decentralized Executive Control in Distributed Computer Systems.
In *Proceedings of COMPCON 78*, pages 254-258. IEEE, November, 1978.

Abstract

This paper discusses the issues involved in building a real-time control system using a message-directed distributed architecture. We begin with a discussion of the nature of real-time software, including the viability of using hierarchical models to organize the software. Next we discuss some realistic design objectives for a distributed real-time system including fault isolation, independent module verification, contex-independence, decentralized control and partitioned system state. We conclude with some observations concerning the general nature of distributed system software.

[Boebert 78b] Boebert, W. E.; Franta, W. R.; Jensen, E. D.; Kain, R. Y.
Kernel Primitives of the HXDP Executive.
In *Proceedings of COMPCON 78*, pages 595-600. IEEE, November, 1978.

Abstract

This paper describes the kernel of an Executive being implemented for the Honeywelll Experimental Distributed Processor (HXDP) -- a vehicle for research in distributed computers for real-time control. The kernel provides message transmission primitives for use by application programs or higher level executive functions. In the paper we describe the message transmission primitives provided by the kernel and the rationale for their selection based upon the objectives and constraints described in a companion paper.

[Boebert XX] W. E. Boebert, D. Cornhill, W. R. Franta, and E. D. Jensen.
Communications in the HXDP Executive.
IEEE Transactions on Software Engineering, 19XX.
to appear.

[Boyd 83] Boyd, R. T.; Dickerson, K. R.; Sager, J. C.
A Distributed Operating System for Reliable Telecommunications Control.
In *Fifth International Conference on Software Engineering for Telecommunication Switching Systems (Lund, Sweden)*, pages 190-195. IEE, July 4-8, 1983.

Abstract

The system consists of a number of loosely-coupled processor modules attached to external hardware. The software is composed of communicating processes. A key design feature is system-wide reconfigurability under operating system control. This means that processes can be allocated to, and migrated between, processor modules as required; for example, following module hardware failures, or for changing workload requirements. Sections outline the processor system architecture and operating system control of communication, configuration management and fault recovery.

[Bruins 83] Bruins, Th.; Vree, W.; Reijns, G.; van Spronsen, C.
A Layered Distributed Operating System.
In *Local Networks. Strategy and Systems. LOCALNET '83 (London, England)*, pages 351-371. March 8-10, 1983.

Abstract

The rapidly decreasing prices and increasing performance of micro electronics, together with the promising developments in the area of digital transmission permit a new approach in distributed computing architecture. The basic aim was to allow a number of micro processors to achieve a common task, behaving toward the user as one abstract machine. In order to avoid vulnerable or critical elements, distribution of tasks has been accomplished in such a manner, that elimination of a processor only degrades but never stops the total service. Special attention is given to the principles of true distributed and parallel processing and the consequences for the operating system services. Emphasis has been put on the description of functions in higher layers such as the call of not locally available functions, the distributed directories and the way they are incorporated and updated. Furthermore, a description is given of the way connection-less data communications has been facilitated in incorporating a storage function at the transport level.

[Carulli 82] Carulli, M.; Murro, O.
Software Architecture of a Locally Distributed System Supporting Network Transparent Applications.
In *Wescon/82 Conference Record (Anaheim CA)*, pages 24-32. Electronics Conventions, September 14-16, 1982.

Abstract

Presents an integrated, distributed system based on an Ethernet network of the Olivetti BCS 2000 system. A fundamental objective of this system is to develop distributed applications at the same level of difficulty as in individual machines. The authors present, in particular, the architecture of the BCOS-M distributed operating system, the design of which is determined by the objectives of network transparency as well as by the needs of resource distribution, reliability and availability.

[Cheriton 79] D. R. Cheriton, M. A. Malcolm, et al.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, February, 1979.

Abstract

This paper describes a portable real-time operating system called Thoth which has been developed at the University of Waterloo as part of a research study into the feassibility of portable operating systems. Thoth supports multiple processes, dynamic memory allocation, device-independent input/output, a file system, multiple terminals, and swappping. It is currently running on two minicomputers with quite different architectures (Texas Instruments 990 and Data General Nova).

[Coleman 79] Coleman, Aaron Ray.
Security Kernel Design for a Microprocessor-Based Multilevel Archival Storage System.
Master's thesis, Naval Postgraduate School, Monterey, CA, December, 1979.

Abstract

This thesis is a detailed design of a security kernel for an archival file storage system. Microprocessor technology is used to address a major part of the problem of information security in a distributed computer system. Utilizing multi-programming techniques for processor efficiency, segmentation for controlled sharing, and a loop-free structures for avoiding intermodule dependencies, the Archival Storage is designed for implementation on the Zilog Z9001 microprocessor with a memory management unit. The concepts of a process structure and a distributed kernel are used in providing management of the shared hardware resources of the system. The security kernel primitives create a virtual machine environment and provide information security in accordance with a non-discretionary security policy.

[Cornhill 79] Cornhill, D. T.; Boebert, W. E.
Implementation of the HXDP Executive.
In *Proceedings of COMPCON 79*, pages 219-221. IEEE, February, 1979.

Abstract

This paper describes a first implementation of the executive for the Honeywell Experimental Disbributed Processor (HXDP). HXDP has been built to investigate distributed, decentralized control in real time applications. The purpose of the implementation is to demonstrate the utility of, and to gain experience with the executive primitives in the area of interprocess communication.

[Czaplicki 81] Czaplicki, C. S.
Advanced Airborne Executive.
In *Sixth Conference on Local Computer Networks (Minneapolis MN)*, pages 10-12. IEEE, October 12-14, 1981.

Abstract

The main objective of this program was to postulate, implement and test a distributed executive design which would meet the requirements of various avionics distributed processing configurations. Future project requirements are reviewed and a distributed processing architecture which best meets the near-term future Navy avionic requirements has been selected. The goal was a general purpose executive program which would provide increased reliability, graceful degradation and expanded processing capability while providing flexibility in architectural design of the configuration of computers and processing functions within a system.

[Finkel 80] Finkel, Raphael; Solomon, Marvin; Tischler, Ron.
Arachne User Guide, Version 1.2.
Technical Report MRC-TSR-2066, Mathematics Research Center, Wisconsin University, Madison, WI, April, 1980.

Abstract

Arachne is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes Arachne from the viewpoint of a user or a writer of user-level programs. All system service calls and library routines are described in detail. In addition, the command-line interpreter and terminal input conventions are discussed. Companion reports describe the purpose and concepts underlying the Arachne project and give detailed accounts of the Arachne utility kernel and utility processes.

[Friedrich 83] Friedrich, G. R.; Eser, F. W.
Management Units and Interprocess Communication of DINOS.
Siemens Forsch.- and Entwicklungsber. (Germany) 12(1):21-27, January, 1983.

Abstract

The structure and implementation aspects of the processing management and the interprocess communication (IPC) offered by DINOS are described. Hierarchically structured software units (execution unit, distribution unit, process) are the basic objects of the processing management. All the software is distributed over a number of independent execution units consisting of a variable number of distribution units. The processing management allocates these software units in a distributed and decentralized way at runtime. The interprocess communication is based on messages. Hierarchical names ensure independence of the IPC interface from the process allocation since IPC data are strictly partitioned and distributed. IPC control and data are completely distributed.

[Fundis 80] Fundis, Roxanna; Wallentine, Virgil.
Command Processors for Dynamic Control of Software Configurations.
Technical Report TR-80-02, Department of Computer Science, Kansas State University, Manhattan, KA, July, 1980.

Abstract

Command language facilities for the construction and execution of software configuration--networks of communicating processes--are very limited today because current operating systems do not support this level of complexity. The Network Adaptable Executive (NADEX) is an operating system which was designed to support dynamic configurations--those configurations which are constructed at command interpretation time--of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. Three command processors have been developed to demonstrate the sufficiency of the NADEX facilities to support dynamic configurations. NADEX facilities, an overview of the Job Control System, and the command processor configuration environment are presented, followed by user's guides for the command processors. Each command processor has different responsibilities and capabilities for handling configurations. The NADEX Static command processor executes completely connected configurations. The UNIX command processor allows linear configurations to be constructed dynamically, and the MIRACLE command processor allows the dynamic construction of arbitrary configurations. Syntax graphs and sample user sessions are presented for each command processor.

[Gatefait 81] Gatefait, J. P.; Surleau, P.; Konrat, J. L.
Execution Mechanisms for Administration Programs in the E10.S System.
In *IEE Fourth International Conference on Software Engineering for Telecommunication Switching Systems (Coventry, England)*, pages 130-137. IEE, July 20-24, 1981.

Abstract

Addresses some of the specific OANDM software problems encountered with a distributed control system like the E10.S, and the solutions adopted. The authors successively discuss: the system's distributed control architecture; the role of the Operator Command Servicing (OCS) programs; some aspects of man-machine communications and OCS program execution; mechanisms for access to system data, and use of a logical model to provide uniform descriptions for all data accessible by operators.

[Geitz 81] Geitz, G. W.; Schmitter, E. J.
BFS-Realization of a Fault-Tolerant Architecture.
In *Eighth Annual Symposium on Computer Architecture (Minneapolis MN)*, pages 163-170. IEEE, ACM, May 12-14, 1981.

Abstract

Considers possibilities of distributed architecture to improve the reliability of microcomputer systems to realize a fault-tolerant system. By using and extending existing redundancies of hardware, software, and time, a partially meshed ring structure that meets the requirements of a fault-tolerant architecture has been designed. Aspects of hardware implementation, system software structure, operating system requirements, fault diagnosis, and reconfiguration are explained, based on the fault-tolerant architecture Basic Fault-tolerant System BFS.

[Glorieux 81] Glorieux, A. M.; Rolin, P.; Sedillot, S.
User Services Offered by the Application Protocol Implemented in SIRIUS-DELTA.
In *Networks from the User's Point of View. Proceedings of the IFIP TC-6 Working Conference COMNET '81 (Budapest, Hungary)*, pages 107-115. IFIP, May 11-15, 1981.

Abstract

Over the years the need for handling distributed applications has increased tremendously. The authors describe the goals and the architecture of the distributed data base management system SIRIUS-DELTA. The attribution of each layer and the protocols are discussed. The user point of view guides the authors in all these definitions. Issues in query decomposition, concurrency control, failure survival, distributed executive, checkpoints and performances evaluations are studied.

[Guillemont 82] Guillemont, M.
The CHORUS Distributed Operating System: Design and Implementation.
In *Local Computer Networks. Proceedings of the IFIP TC 6 International In-Depth Symposium on Local Computer Networks (Florence, Italy)*, pages 207-223. IFIP, April 19-21, 1982.

Abstract

CHORUS is an architecture for distributed systems. It includes a method for designing a distributed application. A structure for it's execution and the (operating) system to support this execution. One important characteristic of CHORUS is that the major part of the system is built with the same architecture as applications. In particular, the exchange of messages, which is the fundamental communication/synchronization mechanism, has been extended to the most basic functions of the system.

[Heger 81] Heger, Dirk.
Completion and Pilot Testing of a Fault Tolerant Real Time Computer System with Distributed Microcomputers: Pilot Implementation (Really Distributed Control (RDC) System).
Technical Report BMFT-FB-DV-81-007, Bundesministerium fuer Forschung und Technologie, Bonn-Bad Godesberg, Germany, December, 1981.

Abstract

A prototype RDC system was tested and the completeness of the hardware and software components were proven in practice by a pilot implementation. Features of the system include: distributed fault tolerant real time computer system with a fiber optic ring-bus system for industrial automation; modular design, central operating by means of an input-output color screen system; and complete programming by means of a multicomputer PEARL. A stepwise upgraded pit furnace plant with 28 pit furnaces was selected as the pilot project. Experience was given in the following areas; reliability fault diagnosis; fault tolerance; fiber optics under environmental stress; traffic flow in the ring-bus system with decentralized control; digital drive and control of a real pit furnace process using the high level language PEARL; synchronization and interprocess communication with PEARL; use of a dynamic down line loader; application of a distributed operating system supporting multicomputer PEARL; adaptation of the PEARL operating system to other computers; and distributed real time data bases.

[Hsia 79] P. Hsia.
A Configurable Distributed Computing System.
In *Proceedings of the First International Conference on Distributed Computing Systems*, IEEE, November, 1979.

[Jensen 78] Jensen, E. D.
The Honeywell Experimental Distributed Processor -- An Overview.
IEEE Computer 11(1):28-38, January, 1978.

Abstract

The Honeywell Experimental Distributed Processor (HXDP) is a vehicle for research in the science and engineering of processor interconnection, executive control, and user software for a certain class of multiple-processor computers which we call 'distributed computer' systems. Such systems are very unconventional in that they accomplish total system-wide executive control in the absence of any centralized procedure, data, or hardware. The primary benefits sought by this research are improvements over more conventional architectures

(such a multiprocessors and computer networks) in extensibility, integrity, and performance. A fundamental thesis of the HXDP project is that the benefits and cost-effectiveness of distributed computer systems depend on the judicious use of hardware to control software costs.

[Jensen 81] E. Douglas Jensen.
Distributed Control,
In B. W. Lampson, M. Paul, and H. J. Siegert, *Distributed Systems - Architecture and Implementation*, pages 175-190. Springer-Verlag, 1981.

[Jessop 82] Jessop, W. H.; Noe, J. D.; Jacobson, D. M.; Baer, J. L.; Pu, C.
The Eden Transaction-Based File System.
In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems (Pittsburgh PA)*, pages 163-169. IEEE, July 19-21, 1982.

Abstract

THE Eden file system employs an object model approach in the design of a transaction-based file system to be used in the Eden distributed system. The file system relies on a kernel which provides both an object model abstraction and a relatively high-level storage system. The Eden file system willl provide all of the functions of a conventional file system. In addition, it will serve as a research tool-kit, both for developing distributed applications which depend on a general transaction mechanism and for research into the performance of different concurrency control methods which can be used within the transaction mechanism.

[Jones 79] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings of the Symposium on Operating Systems Principles*, ACM, December, 1979.

[Karshmer 83] Karshmer, A. I.; Phelan, J.; Kempton, B.; Depree, D. J.
The New Mexico State University Distributed UNIX System: Evaluation and Extension.
In *Proceedings of the Sixteenth Hawaii International Conference on System Sciences (Honolulu HI)*, pages 225-233. University of Hawaii, University of Southwestern Louisiana, January 5-7, 1983.

Abstract

Through a joint effort between New Mexico State University and the Hebrew University of Jerusalem, a distributed version of the UNIX operating system is currently being developed. A microprocessor version of the UNIX kernel has been designed and implemented to run on any member of the PDP-11/LSI-11 family of processors and allows programs to run in a 'UNIX-like' environment. As the kernels running in the distributed processing elements present a 'UNIX-like' environment, all processes in the system are fully transportable from one processor to another. While the original version of the system was built in a star configuration, the system is currently being enhanced through the addition of a communication ring which uses 8-bit microprocessors as ring interface units. The paper describes microprocessors as ring interface units. The paper describes the software and hardware structure of the system as well as some performance measurements taken on the basic star version of the implementation.

[Kartashev 82] Kartashev, S. I.; Kartashev, S. P.
A Distributed Operating System for a Powerful System with Dynamic Architecture.
In *AFIPS Conference Proceedings, Vol 51, 1982 National Computer Conference (Houston TX)*, pages 103-116. AFIPS, June 7-10, 1982.

Abstract

The paper discusses the organization of a distributed operating system for dynamic architecture. It is shown that the operating system must feature two types of distribution: (A) functional or vertical, wherely it is distributed among functional units in accordance with the types of conflicts that should be resolved; and (B) modular or horizontal, whereby it is distributed among modules performing the same functions. In a dynamic architecture there are three types of conflicts; memory, reconfiguration, and I/O/ This leads to the division of OS into three subsystems: (1) a processor OS that resolves memory conflicts, (2) a monitor OS that resolves reconfiguration conflicts, and (3) an I/O OS that resolves all types of I/O conflicts. The paper presents a detailed organization for the processor operating system.

[Kieburtz 81] Kieburtz, R. B.
A Distributed Operating System for the Stony Brook Multicomputer.
In *Second International Conference on Distributed Computing Systems (Paris, France)*, pages 67-79. Inst. Nat. Recherche and Inf. Autom.; Lab. Recherche and Inf.; Paris-Sud University of Orsay, April 8-10, 1981.

Abstract
The Stony Brook multicomputer is a hierarchically organized network of computer nodes that has been designed to support problem-solving by decomposition. High performance, relative to the speed of its individual processors, is one of its primary design goals. This paper describes the design of a message-based, distributed, operating system nucleus for the network.

[Lacoss 80] Lacoss, Richard T.
Distributed Sensor Networks.
Technical Report ESD-TR-80-244, Electronic Systems Division, Hanscom AFB, MA, September, 1980.

Abstract
This Semiannual Technical Summary reports work in the Distributed Sensor Networks program for the period 1 April through 30 September 1980. Progress related to development and deployment of test-bed hardware and software, including deployment of three test-bed nodes, is described. A complete algorithm chain from raw data to aircraft locations, employing two acoustic arrays, has been developed and demonstrated experimentally using data collected from test-bed nodes. A strawman design for a new multiple microprocessor test-bed node computer is presented. Also described is progress in the design and development of a real-time network kernel for the DSN test bed in general, and the new processor in particular.

[Lantz 82] Lantz, K. A.; Gradischnig, K. D.; Feldman, J. A.; Rashid, R. F.
Rochester's Intelligent Gateway.
IEEE Computer 15(10):54-68, October, 1982.

Abstract
The University of Rochester has had several years experience in the design and implementation of a multiple-machine, multiple-network distributed system called RIG, or Rochester's Intelligent Gateway. RIG was designed as a state-of-the-art research computing environment to support a variety of distributed applications and research in distributed computing. Particular applications include computer image analysis and design automation for VLSI. Distributed systems research includes investigations into internetwork architectures, interprocess communication, naming, distributed file systems, distributed control, performance monitoring, exception handling, debugging, and user interfaces.

[Lazowska 81] Lazowska, E. D.; Levy, H. M.; Almes, G. T.; Fischer, M. J.; Fowler, R. J.; Vestal, S. C.
The Architecture of the Eden System.
Operating Systems Review 15(5):148-159, December, 1981.

Abstract
The University of Washington's EDEN project is a five-year research effort to design, build and use an integrated distributed computing environment. The underlying philosophy of Eden involves a fresh approach to the tension between these two adjectives. In briefest form, Eden attempts to support both good personal computing and good multi-user integration by combining a node machine/local network hardware base with a software environment that encourages a high degree of sharing and cooperation among its users. The hardware architecture of Eden involves an Ethernet local area network interconnecting a number of node machines with bit-map displays, based upon the INTEL IAPX 432 processor. The software architecture is object-based, allowing each user access to the information and resources of the entire system through a simple interface. This paper states the philosophy and goals of Eden, describes the programming methodology that has been chosen to support, and discusses the hardware and kernel architecture of the system.

[LeLann 81] LeLann, G.
A Distributed System for Real-Time Transaction Processing.
IEEE Computer 14(2):43-48, February, 1981.

Abstract
The computing systems considered in this article are built from a variety of commonly available hardware components for processing, storage, and communication, such as minicomputers, disks, and buses. Physically

distributed over short distances, these systems are usually labeled as multiple-processor computers or local area computer networks. We begin by outlining the basic problems that were addressed during the design of Delta, an experimental distributed transactional system built within the framework of Project Sirius. We then discuss some of the advantages of distributed architectures and conclude with a presentation of the basic aspects of Delta's distributed executive mechanisms.

[Liu 82] Ming T. Liu; Duen-Ping Tsay; Lian, R. C.
Design of a Network Operating System for the Distributed Double-Loop Computer Network (DDLCN).
In *Local Computer Networks. Proceedings of the IFIP TC 6 International In-Depth Symposium on Local Computer Networks (Florence, Italy)*, pages 225-248. IFIP, April 19-21, 1982.

Abstract
Presents the framework and model of a Network Operating System (NOS) for use in distributed systems in general and for use in the distributed double-loop computer network (DDLCN) in particular. An integrated approach is taken to design the NOS model and protocol structure. It is based on the object model and a novel 'task' concept, using message passing as an underlying semantic structure. A layered protocol is provided for the distributed system kernel to support NOS. This approach provides a flexible organization in which system-transparent resource sharing and distributed computing can evolve in a modular fashion.

[Luderer 81] Luderer, G. W. R.; Che, H.; Haggerty, J. P.; Kirslis, P. A.; Marshall, W. T.
A Distributed UNIX System Based on a Virtual Circuit Switch.
Operating Systems Review 15(5):160-168, December, 1981.

Abstract
The popular UNIX operating system provides time-sharing service on a single computer. This paper reports on the design and implementation of a distributed UNIX system. The new operating system consists of two components: The S-UNIX subsystem provides a complete UNIX process environment enhanced by access to remote files; the F-UNIX subsystem is specialized to offer remote file service. A system can by configured out of many computers which operate either under the S-UNIX or the F-UNIX operating subsystems. Computers communication with each other through a high-bandwidth virtual circuit switch. Small front-end processors handle the data and control protocol for error and flow-controlled virtual circuits. Terminals may be connected directly to the computers or through the switch. Operational since early 1980, the system has served as a vehicle to explore virtual circuit switching as the basis for distributed system design. The performance of the communication software has been a focus of the work. Performance measurement results are presented for user process level and operating system driver level data transfer rates, message exchange times, and system capacity benchmarks. The architecture offers reliability and modularly growable configurations. The communication service offered can serve as a foundation for different distributed architectures.

[Lycklama 78] Lycklama, H.; Bayer, D. L.
The MERT Operating System.
The Bell System Technical Journal 57(6):2049-2086, July, August, 1978.

Abstract
The MERT operating system supports multiple operating system environments. Messages provide the major means of inter-process communication. Shared memory is used where tighter coupling between processes is desired. The file system was designed with real-time response being a major concern. The system has been implemented on the DEC PDP-11/45 and PDP-11/70 computers and supports the UNIX time-sharing system, as well as some real-time processes. To provide an environment favorable to applications with real-time response requirements, the MERT system permits processes to control scheduling parameters. These include scheduling priority and memory residency. A rich set of inter-process communication mechanisms including messages, events (software interrupts), shared memory, inter-process traps, process ports, and files, allow applications to be implemented as several independent, cooperating processes. Some uses of the MERT operating system are discusses. A retrospective view of the MERT system is also offered. This includes a critical evaluation of some of the design decisions and a discussion of design improvements which could have been made to improve overall efficiency.

[Mahjoub 82] Mahjoub, A.
A Distributed Operating System for a Local Area Network.
In *Ninth Australian Computer Conference Vol. 2 (Hobart, Tasmania, Australia)*, pages 633-647. August 23-27, 1982.

Abstract
The design and implementation of an experimental distributed operating system for a local area network are discussed. The salient feature of this operating system is that it achieves complete machine transparency and atomicity of remote operations. The system, as a whole, provides a suitable environment for a distributed version of the concurrent programming language MODULA without introducing any modification to its compiler.

[Maisonneuve 81] Maisonneuve, M.; Levy, J. P.; Konrat, J. L.
E10.S Operating System for a Distributed Architecture.
In *IEE Fourth International Conference on Software Engineering for Telecommunication Switching Systems (Coventry, England)*, pages 124-129. IEE, July 20-24, 1981.

Abstract
Describes the general structure of computer-controlled telephone exchanges and rather briefly discusses E10.S hardware, before entering into the details of the system's software, the main subject of this paper.

[Mamrak 83] Mamrak, S. A.; Leinbaugh, D.; Berk, T. S.
A Progress Report on the Desperanto Research Project: Software Support for Distributed Processing.
Operating Systems Review 17(1):17-29, January, 1983.

Abstract
The DESPERANTO research project has been investigating topics in the area of distributed computing systems since the fall of 1980. The project addresses problems that arise in the design and implementation of software support for general-purpose resource sharing in networks consisting of heterogeneous nodes. Aothough it is still premature to publish the details of the solutions to the desgn problems in journal (or archival) form, this report has been prepared to describe design issues and progress made to date.

[Manning 77] Manning E.; Peebles R. W.
A Homogeneous Network for Data Sharing - Communications.
Computer Networks 1(4):211-224, June, 1977.

[McCarthy 81] McCartny, J. L.; Merrill, D. W.; Marcus, A.; Benson, W. H.; Gey, F. C.
SEEDIS Project: A Summary Overview.
Technical Report PUB-424, Department of Energy, Washington, DC (UC-Berkeley), September, 1981.

Abstract
The SEEDIS project includes: a research program to investigate information systems spanning diverse data sources, computer hardware and operating systems; a testbed distributed information system running on a network of Digital Equipment Corporation (DEC) VAX computers, which is used for selected applications as well as research and development; a set of interactive information management and analysis tools in fields such as energy and resource planning, employment and training program management, and environmental epidemiology; and a major collection of databases for various geographic levels and time periods drawn from the US Census Bureau and other sources.

[McDonald 82] McDonald, W. C.; Smith, R. W.
A Flexible Distributed Testbed for Real-Time Applications.
IEEE Computer 15(10):25-38, October, 1982.

Abstract
This article describes a flexible distributed testbed that is being developed to support the development, analysis, test, evaluation, and validation of research in distributed computing for real-time applications. The testbed not only provides the resources for experimentally obtaining quantitative results, but also sserves as a focal point for the research, integrating related research activities and providing a mechanism for technology transfer to associated research efforts.

[McKendry 83] McKendry, M. S.; Allchin, J. E.; Thibault, W. C.
Architecture for a Global Operating System.
In *Proceedings of IEEE INFOCOM 83 (San Diego, CA)*, pages 25-30. IEEE, April 18-21, 1983.

Abstract

Global operating systems are suited to distributed, local-area network environments. A decentralized global operating system can manage all resources globally, relying on functional requirements for resource allocation, rather than on the relative physical locations of the resource allocation mechanism and the resource itself. Among the advantages of global operating systems are the ability to use idle resources and to control the environment as a single cohesive entity. This paper introduces an architectural approach to supporting decentralized global operating systems. The approach addresses the problem of managing distributed data by incorporating specialised data management facilities in the kernel. This data management is especially useful to the operating system itself. A capability-based access scheme provides flexible control of resources and autonomy. The approach is being utilised in the Clouds operating system project at Georgia Institute of Technology.

[Measures 82] Measures, M.; Carr, P. A.; Shriver, B. D.
A Distributed Operating System Kernel Based on Dataflow Principles.
In *Proceedings of Computer Networks COMPCON 82. Twenty-fifth IEEE Computer Society International Conference (Washington DC)*, pages 106-115. IEEE, September 20-23, 1982.

Abstract

The design of the Distributed Operating System Kernel, or DOSK, is presented as an operating system for a distributed computing system. An extended dataflow model forms the basis for both the programs DOSK executes and the implementation of DOSK itself. DOSK can realize the parallelism in a program by distributing portions of the program across the system for concurrent execution. DOSK consists of several asynchronous processes that communicate via message-passing using a dataflow protocol.

[Miller 81] Miller, B.; Presotto, D.
XOS: An Operating System for the X-tree Architecture.
Operating Systems Review 15(2):21-32, April, 1981.

Abstract

Describes the fundamentals of the X-tree Operating System (XOS), a system developed to investigate the effects of the X-tree architecture on operating system design. It outlines the goals and constraints of the project and describes the major features and modules of XOS. Two concepts are of special interest: the first is demand paging across the network of nodes and the second is separation of the global object space and the directory structure used to reference it. Weaknesses in the model are discussed along with directions for future research.

[Miller 83] Miller, D. S.; Fisher, R. W.; Millard, B. R.; Murthy, V. G.
A Distributed Operating System for a Local Area Network.
In *Second Annual Phoenix Conference on Computers and Communications. 1983 Conference Proceedings (Phoenix AZ)*, pages 281-288. IEEE, March 14-16, 1983.

Abstract

HERBERT-II is a distributed operating system which runs on a local area network of three 6809 based codex intelligent terminal system computers fully connected by MC6821 PIA parallel interfaces. The codex ISOS operating system at each node has been extended to include physical, link, network, transport and session communication layers normally added on as an afterthought in access methods or utilities in conventional distributed system architectures. HERBERT-II is a object-oriented UNIX-like operating system which supports multiprogramming on multiple processors.

[Muntz 83] Muntz, Charles A.
NSW (National Software Works) Executive Enhancements II.
Technical Report RADC-TR-83-59, Rome Air Development Center, Griffiss AFB, NY, March, 1983.

Abstract

The National Software Works (NSW) represents a significant evolutionary in the fields of distributed processing and network operating systems. Its ambitious goal has been to link the resources of a set of geographically distributed and heterogeneous hosts with an operating system which would appear as a single entity to a user. It is principally aimed at the development of software systems and at providing software tools which can be used to support the software development activity throughout its life cycle. This report describes the current status of the NSW system as well as highlights the enchancements and improvements made to the NSW system during the past two years.

[Ousterhout 80] Ousterhout J. K.;Scelza D. A.;Sindhu P. S.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.

Abstract

The design of Medusa, a distributed operating system for the Cm* multimicroprocessor, is discussed. The Cm* architecture combines distribution and sharing in a way that strongly impacts the organization of operating systems. Medusa is an attempt to capitalize on the architectural features to produce a system that is modular, robust, and efficient. To provide modularity and to make effective use of the distributed hardware, the operating system is partitioned into several disjoint *utilities* that communmicate with each other via messages. To take advantage of the parallelism present in Cm* and to provide robustness, all programs, including the utilities, are *task forces* containing many concurrent, cooperating *activities*.

[Popek 81] Popek, G.; Walker, B.; Chow, J.; Edwards, D.; Kline, C.; Rudisin, G.; Thiel, G.
LOCUS: A Network Transparent, High Reliability Distributed System.
Operating Systems Review 15(5):169-177, December, 1981.

Abstract

LOCUS is a distributed operating system that provides a very high degree of network transparency while at the same time supporting high performance and automatic replication of storage. By network transparency the authors mean that at the system call interface there is no need to mention anything network related. Knowledge of the network and code to interact with foreign sites is below this interface and is thus hidden from both users and programs under normal conditions. LOCUS is application code compatible with UNIX, and performance compares favorably with standard, single system UNIX. LOCUS runs on a high bandwidth, low delay local network. It is designed to permit both a significant degree of local autonomy for each site in the network while still providing a network-wide, location independent name structure. Atomic file operations and extensive synchronization are supported. Small, slow sites without local mass store can coexist in the same network with much larger and more powerful machines without larger machines being slowed down through forced interaction with slower ones. Graceful operation during network topology changes is supported.

[Rapantzikos 81] Rapantzikos, Demosthenis K.
Detailed Design and Implementation of the Kernel of a Real-Time Distributed Multiprocessor Operating System.
Master's thesis, Naval Postgraduate School, Monterey, CA, March, 1981.

Abstract

This thesis presents the detailed design and implementation of the kernel of a real-time, distributed operating system for a microcomputer based multiprocessor system. Process oriented structure, segmented address spaces and a synchronization mechanism based on event counts and sequencers comprise the central concepts around which this operating system is built. The operating system is hierarchically structured, layered in three loop free levels of abstraction and fundamentally configuration independent. This design permits the logical distribution of the kernel functions in the address space of each process and the physical distribution of system code and data among the microcomputers. This physical distribution in turn, in a multimicroprocessor configuration will help to minimize system bus contention. The system particularly supports applications where processing for which this system has been specifically developed. The implementation was developed for the INTEL 86/12A single-board computer using the 8086 processor chip.

[Rashid 81] Rashid, R. F.; Robertson, G. G.
Accent: A Communication Oriented Network Operating System Kernel.
Operating Systems Review 15(5):64-75, December, 1981.

Abstract

Accent is a communication oriented operating system kernel being built at Carnegie-Mellon University to support the distributed personal computing project, SPICE, and the development of a fault-tolerant Distributed Sensor Network (DSN). Accent is built around a single, powerful abstraction of communication between processes, with all kernel functions, such as device access and virtual memory management accessible through messages and distributable throughout a network. In this paper, specific attention is given to system supplied facilities which support transparent network access and fault-tolerant behavior. Many of these facilities are already being provided under a modified version of VAX/UNIX. The Accent system itself is currently being implemented on the Three Rivers Corp. PERQ.

[Reiter 81] Reiter, E. E.; Zimmerman, D. L.
Distributed Operating System for Cooperating Functional Processors.
Technical Report UCID-19847, Lawrence Livermore National Laboratory, CA, 1981.

Abstract

The paper has been divided into several main chapters. This first chapter contains a discussion of the goals of the system, the architecture assumptions used, and the structure of FP. The next chapters present an overview and discussion of the service support layer processes, the other upper level supports for the partition of problems, parallel processing, and the implementation of the FP language. These chapters are the heart of the paper, in the sense that they deal with the possibility of an implementation of a parallel processor that accepts FP. The next chapter is a discussion of the kernel of the operating system. In this distributed system, this is the message passing system. It allows processes on the same or different nodes to communicate, and is thus the backbone of the entire system. Finally, we have included another chapter which reviews some of the issues covered in this system. For instance, we included discussions of synchronization, resource allocation, and protection.

[Restorick 82] Restorick, F. M.; Pardoe, B. H.
A Multi-Microprocessor Design for Use in a Packet Switched Network.
In *Pathways to the Information Society. Proceedings of the Sixth International Conference on Computer Communication (London, England)*, pages 811-816.
International Council of Computer Communication, September 7-10, 1982.

Abstract

Describes a multi-processor architecture which is particularly suited to act as a node processor in a packet switching environment. The basic concept is that each high speed link entering the node has its own dedicated module, containing its own packet memory, CPU, and operating software. There are two global busses which act as an interconnect between the separate modules. These are the system bus, and an inter CPU bus. Due to the 'loose' coupling between each processor module, the possibility of failure of the whole node is reduced. The basic kernel of the distributed operating system needed to run this multi-processor as a packet switching node is discussed. The recovery mechanisms with regard to a link module failure is also dealt with.

[Rieger 79] Rieger, Chuck.
ZMOB: A Mob of 256 Cooperative Z80A-Based Microcomputers.
Technical Report TR-825, Department of Computer Science, Maryland University, College Park, MD, November, 1979.

Abstract

Current directions of computer science and computing in general are toward more parallel machine architectures and distributed models of computing based upon these new architectures. Recently, there has been considerable interest in highly parallel architectures capable of supporting complex distributed computation via a large number of autonomous processors. ZMOB is such a machine, currently under design and simulation. Architecturally, ZMOB is a collection of 256 identical but autonomous Z80A-based microcomputers (processors). Each processor comprises 32K bytes of 375 ns read/write central memory (expandable to 48K bytes), up to 4K bytes of resident operating system on 450ns EPROM, an 8-bit hardware multiplier, and interface logic for communications functions.

[Rieger 81] Rieger, C.
ZMOB: Doing it in Parallel!
In *1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management (Hot Springs VA)*, pages 133-140.
IEEE, November 11-13, 1981.

Abstract

The architecture and applications of ZMOB, a 256 processor computer for artificial intelligence and general computer science research, are described. This machines's 16 million byte distributed memory, 100 million instruction per second overall throughput, and high speed interprocessor communication make ZMOB attractive and appropriate for a wide range of basic and applied research in parallel computing. ZMOB's price tag is approximately $150K, and the machine will be operational by late 1981.

[Rivoira 82] Rivoira, S.; Serra, A.
A Multimicro Architecture and its Distributed Operating System for Real Time Control.
In *Proceedings of the 3rd International Conference on Distributed Computing Systems (Miami/Fort Lauderdale FL)*, pages 238-246. IEEE, October 18-22, 1982.

Abstract

In a tightly coupled multi-microcomputer system suitable for process control applications, the microcomputers are grouped into a cluster and communicate using a high speed parallel common bus. Hardware mechanisms are provided as supports for the implementation of synchronization primitives between processes allocated on different processors. The system fault-tolerance is achieved by memory management units, which relocate and protect programs and data against faults and programming mistakes. The distributed operating system kernel makes available a virtual machine where processes allocated on different processors are executed in parallel, and processes which reside on the same processor are executed in a multitasking environment.

[Schmidtke 82] Schmidtke, F. E.
A Communication Oriented Operating System Kernel for a Fully Distributed Architecture.
In *Pathways to the Information Society. Proceedings of the Sixth International Conference on Computer Communication (London, England)*, pages 757-762. International Council of Computer Communication, September 7-10, 1982.

Abstract

Starting with a description of the considered network architecture of the loosely coupled multimicrocomputer system SIELOCNET. The basic design principles of the approach are outlined. The currently implemented network operating system called DINOS is based on autonomous system software for all computer nodes which cooperate with other components by well defined protocols. It is based on a state-of-the-art realtime-multitasking kernel managing the local activities of a single node. The DINOS communication mechanism across computer boundaries as well as the overall load balancing and allocation management are embedded within a layered structure of each local operating system. For a programmer there is a unique addressing scheme for local objects within a single computer and remote ones residing elsewhere.

[Schmidtke 83] Schmidtke, F. E.
Operating System for an Optical-Bus Local Network.
Siemens Forsch.- and Entwicklungsber. (Germany) 12(1):16-20, January, 1983.

Abstract

The report introduces the network architecture of SIELOCNET and its functional decomposition into workstations, dedicated computers and arbitrary processing nodes. The basic design goals and characteristics of DINOS, a Distributed Network Operating System, are outlined. It is designed and implemented as a hierarchically layered system providing a separation of mechanisms and strategies and offering a completely transparent interface to individual application. DINOS consists of a collection of autonomous but cooperative local node operating systems, each of which is a collection of partly replicated, partly specific software modules bound together in a system generation procedure. Together they define the functional capabilities of a node.

[Sedillot 80] S. Sedillot and G. Sergeant.
The Consistency and Execution Control Systems for a Distributed Data Base in SIRIUS-DELTA.
Paper proposed to IFIP 80 Congress.

[Sergeant 79] G. Sergeant and L. Treille.
SER: A System for Distributed Execution Based on Decentralized Control Techniques.
Paper proposed to IFIP 80 Congress.

[Solomon 79] M.H. Solomon and R.A. Finkel.
The Roscoe Distributed Operating System.
In *Proceedings 7th ACM Symposium of Operating Systems Principles*, pages 108-114. ACM, December, 1979.

[Springer 82] Springer, J. F.
The Architecture of a Multi-computer Signal Processing System.
In *Proceedings of the Real-Time Systems Symposium (Los Angeles, CA)*, IEEE, December 7-9, 1982.

Abstract

This paper describes the architecture of a recently developed multi-microcomputer signal processor. The purpose of this development is to provide a flexible system capable of ready application to a variety of signal processing problems using a combination of special purpose and off-the-shelf single board computers. The system is supported by an equally flexible distributed software system comprising operating systems and application components.

[Tanenbaum 81] Tanenbaum, A. S.; Mullender, S. J.
An Overview of the Amoeba Distributed Operating System.
Operating Systems Review 15(3):51-64, July, 1981.

Abstract

Describes the design of a distributed operating system, AMOEBA, intended to control a collection of machines based on the pool-of-processors idea.

[Tokuda 83] Hideyuki Tokuda, Sanjay R. Radia and Eric G. Manning.
Shoshin OS: a Message-based Operating System for a Distributed Software Testbed.
In *Proceedings of the Sixteenth Hawaii International Conference on System Sciences, 1983 (Honolulu HI)*, pages 329-338. University of Hawaii, University of Southwestern Louisiana, January 5-7, 1983.

Abstract

A distributed software testbed, called SHOSHIN, has been constructed to study the development and evaluation of distributed software. The SHOSHIN system consists of two PDP 11/45's and ten LSI 11/23's connected by a tailormade high-speed, parallel bus, called the SCHOOLBUS. The SHOSHIN OS runs on each LSI 11/23 processor, to provide a distributed program environment. This paper describes the software architecture of the SHOSHIN OS, focusing on network transparent process management and interprocess communication.

[Trigg 81] Trigg, R.
Software on ZMOB: An Object-Oriented Approach.
In *1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management (Hot Springs VA)*, pages 133-140. IEEE, November 11-13, 1981.

Abstract

This paper discusses the future of software on ZMOB with particular attention paid to the object-oriented programming style. Included is a look at the current languages supported by ZMOB as well as future possibilities. The suitability of the object-oriented style for ZMOB is discussed and various application areas are briefly described including the domain of mechanism simulation. Finally some ramifications of object-oriented programming to graphics applications are pointed out.

[Tsay 81] Duen-Ping Tsay; Liu, M. T.
MIKE: A Network Operating System for the Distributed Double-Loop Computer Network (DDLCN).
In *Proceedings of COMPSAC 81. IEEE Computer Society's Fifth International Computer Software and Applications Conference (Chicago, IL)*, pages 388-402. IEEE, November 16-20, 1981.

Abstract

This paper presents the framework and model of a network operating system (NOS) called MIKE for use in

distributed systems in general and for use in the distributed double-loop computer network (DDLCN) in particular. MIKE, which stands for Multicomputer Integrated KErnel, provides system-transparent operating for users and maintains cooperative autonomy among local hosts. An integrated approach is taken to design the NOS model and protocol structure. MIKE is based on the object model and a novel 'task' concept, using message passing as an underlying semantic structure. A layered protocol is provided for the distributed system kernel to support NOS. This approach provides a flexible organization in which system-transparent resource sharing and distributed computing can evolve in a modular fashion. In this paper, the NOS model as well as the notion of 'task' are first presented and the system naming convention is then examined. A two-level process interaction model is next described. The protection mechanism is then discussed emphasizing maximal error confinement. A scenario for system-transparent resource sharing using the above concepts is also given. Finally, a multilayer, multidestination protocol structure is detailed.

[Tsuruho 82] Tsuruho, S.; Murata, N.; Haihara, M.
Design and Implementation of DIPS 104-03 Operating System for Distributed Processing.
REVIEW of the Electrical Communication Laboratories 30(6):990-1000, November, 1982.

Abstract
Describes the DIPS distributed processing system design and implementation, and clarifies the software technology in realizing the system. The distributed processing technology is discussed for two cases: the large scale distributed system and load distributed system, as follows: (1) How to share the functions between communication processing and information processing. (2) How to retain the distribution transparency for application program. (3) How to control interprocessor communication. (4) How to manage the files shared among processes.

[Van Den Eijnden 82]
Van Den Eijnden, P. M. C. M.; Dortmans, H. M. J. M.; Kemper, J. P.; Stevens, M. P. J.
Jobhandling in a Network of Distributed Processors.
Technical Report EUT-82-E-131, Eindhoven Univ. Technol., Netherlands, October, 1982.

Abstract
Describes the development of a completely distributed modular computer system. The system is composed of processing units, which can perform specified tasks independently. Adding intelligence to peripheral devices, by means of microprocessors and buffer memories, provides for independent functioning of these peripherals. An intensive transport between the devices is required. The devices are therefore connected by means of a nonblocking communication network, to gain full profit of their intelligence. The intelligent devices are also connected to a central facility containing the operating system. The operating system is distributed over a number of cooperating modules. Each operating system module supports one intelligent device. The operating system modules control the load among the devices and see to the correct processing of jobs, presented by a user. Each is equipped with its own buffer capacities and processing power. The network that interconnects the operating system modules has the same structure as that linking the devices. The operating system modules are relatively simple, because each intelligent device has the same characteristics, seen from the operating system point of view.

[Van Der Linden 81]
Van Der Linden, R.
A Multi-Processor System for Data Communication.
In *Implementing Functions: Microprocessors and Firmware. Seventh Euromicro Symposium on Microprocessing and Microprogramming (Paris, France)*, pages 117-123. September 8-10, 1981.

Abstract
A research project into developing a system for data switching and handling data is described. The system is based on microprocessors supported by large scale integrated peripherals, communicating with each other over a high speed bus. The internal data transmission rate is 10 megabytes. The software of the system is based on the distributed system approach, where a job is performed by several processes. The multitask operating system was especially developed for handling real-time applications and for solving difficulties relevant to the data environment.

[Van Tilborg 81a] VanTilborg, A. M.; Wittie, L. D.
Distributed Task Force Scheduling in Multi-Microcomputer Networks.
In *AFIPS Conference Proceedings (Chicago, IL)*, pages 283-289. AFIPS, May 4-7, 1981.

Abstract

Efficient task scheduling techniques are needed for microcomputer networks to be used as general purpose computers. The wave schuduling technique, developed for the MICRONET network computer, co-schedules groups of related tasks onto available network nodes. Scheduling managers are distributed over a logical control hierarchy. They subdivide requests for groups of free worker nodes and send waves of requests towards the leaves of the control hierarchy, where all workers are located. Because requests from different managers compete for workers, a manager may have to try a few times to schedule a task force, each task force manager actually requests slightly more workers than it really needs. It computes a request size which minimizes expected scheduling overhead, as measured by total idle time in worker nodes, using a Markov queueing model, it is shown that wave scheduling in a network of microcomputers is almost as effecient as centralized scheduling.

[Van Tilborg 81b] Van Tilborg, A. M.; Wittie, L. D.
Wave Scheduling: Distributed Allocation of Task Forces in Network Computers.
In *Second International Conference on Distributed Computing Systems (Paris, France)*, pages 337-347. Inst. Nat. Recherche and Inf. Autom.; Lab. Recherche and Inf.; Paris-Sud University of Orsay, April 8-10, 1981.

Abstract

The new wave scheduling technique is described and analyzed. It distributes task force scheduling by recursively subdividing and issuing wavefront-like requests to worker nodes capable of executing user tasks. The technique is not restricted to any particular network computer interconnection topology. It uses a hierarchical high-level operating system control structure to partition competing task forces among nodes in any network structure. A cost model shows how to minimize wasted processing capacity by using perceived network load to vary the wave scheduling technique.

[Vosbury 82] Vosbury, N.; Bryant, C.
System Software for Experiments in Distributed Computing on a Distributed Testbed.
In *Proceedings of the 3rd International Conference on Distributed Computing Systems (Miami/Fort Lauderdale, FL)*, pages 410-415. IEEE, October 18-22, 1982.

Abstract

Describes the system software for supporting experiemnts in distributed computing on a crossbar-interconnected multi-microprocessor system testbed. This software includes operating system services, system utilities, and a compiler for the language PDL. The PDL compiler includes a type transfer capability, a special procedure call, and utilities for tasking that support operating system work. Operating system components include Nucleus Monitor Services (NMS). The Kernel Operating System (KOS), and the Master Operating System (MOS). NMS provides the most basic services in each microcomputer. KOS executes in each microcomputer and is responsible for managing the local resources. MOS provides global management for the crossbar system computing resources and an interface to an architecture design system that can be used to construct experiments on existing testbed hardware.

[Wasano 81] Wasano, T.; Kamio, M.; Amano, K.
Development of Executive Program in DIPS 104-02 Operating System.
REVIEW of the Electrical Communication Laboratories 29(5-6):368-394, May-June, 1981.

Abstract

Design considerations for the DIPS 104-02 operating system executive program, applied to the large scale data communication systems, are discussed from the following points of view: software layer structure and the functions of each layer; virtualization and distributed processing techniques in the computer center and in the network; and operationability and reliability. New methods to improve control performance for the high speed KANJI printer and the CPU/memory resources scheduling are discussed.

[Wasson 80] Wasson, Warren James.
Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System.
Master's thesis, Naval Postgraduate School, Monterey, CA June, 1980.

Abstract

This thesis describes the detailed design of a distributed operating system for a real-time, microcomputer based multiprocessor system. Process structuring and segmented address spaces comprise the central concepts around which this system is built. The system particularly supports applications where processing is partitioned into a set of multiple processes. One such area is that of digital signal processing for which this system has been specifically developed. The operating system is hierarchically structured to logically distribute its functions in each process. This and loop-free properties of the design allow for the physical distribution of system code and data amongst the microcomputers. In a multiprocessor configuration, this physical distribution minimizes system bus contention and lays the foundation for dynamic reconfiguration.

[Waumans 82] Waumans, B. L. A.
Software Aspects of the Phidias System.
Philips Tech. Rev. (Netherlands) 40(8-9):262-268, August, 1982.

Abstract

The PHIDIAS distributed communication system is built up from 'PRIMEs' (PRocessors with Individual MEmory), which exchange messages by means of a common communication network. The system does not have a common memory. PHIDIAS executes programs that themselves have a distributed character. To enable programs to be written that are independent of a specific architecture or a particular computer system, an existing programming language was extended to include the facility for building up programs from independent processes (called 'SOMAS') that exchange messages with one another. The operating system of PHIDIAS comprises a global operating system and a number of local operating systems for the different PRIMEs. The global operating system can put defective PRIMEs out of action in the event of errors and redistribute the programs among the remaining PRIMEs. The local operating systems ensure that a number of SOMAS can run on one PRIME.

[Waxman 80] Waxman, Robert; Domitz, Robert; Goldberg, Frederick.
Communications Processor Operating System, Volume 8, Task 8, System/Subsystem Specification.
Technical Report RADC-TR-80-187-VOL-8, Plessey, Fairfield, NJ, June, 1980.

Abstract

The Communications Processor Operating System (CPOS) effort is one program of a multiple program effort whose purpose is the development of a Unified Digital Switch (UDS) for strategic communications. This switch will have the capability to perform circuit, packet and store-and-forward message switching in an integrated communication complex. The Communications Processor System (CPS) will control the switching node and will be supported by an operating system called the Communications Processor Operating System. In particular, multilevel communications security conforming to DoD requirements represents a difficult problem for the CPOS and requires solutions which are on the fringe of the current technology. In addition, the need for high reliability is a cause of concern because of the inexact science of software technology. These concerns have resulted in heavy emphasis being given to Tasks 2, 3, 6 and 7. A specification has been prepared as a stand-alone document suitable for the next stage of contractual or in-house development of the CPOS.

[Wilcox 81] Wilcox, Dwight.
Computer Hardware Executive: Concept and Hardware Design.
Technical Report NOSC/TR-721, Naval Ocean Systems Center, San Diego, CA, September, 1981.

Abstract

Large multiprocessing and distributed processing computer systems suffer from diminishing returns in system performance as additional processors are added. The slow execution speed of executive software is one of the principal causes of this phenomenon. The purpose of the executive software is to regulate the time when the various application programs gain access to the computer system resources. This task investigated the potential of special-purpose hardware to eliminate the execution-speed bottlenecks within executive software. A unit, named the Hardware Executive, was designed and fabricated. The Navy standard SDEX/M executive was used as a model. Algorithms were developed for the executive functions of task creation, task dispatching, intratask coordination, real-time clock management, and event-to-task registration and translation.

[Wittie 80] Wittie, L. D.
A Distributed Operating System for a Reconfigurable Network Computer.
IEEE Transactions on Computers, 1980.

[Wittie 82] Wittie, L. D.; Fischer, D. M.
The Design of a Portable Distributed Operating System.
In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences Vol. 1 (Honolulu HI)*, pages 324-332. University of Hawaii, University of Southwestern Louisiana, January 6-8, 1982.

Abstract

MICROS is the distributed operating system for MICRONET, a reconfigurable network of sixteen loosely-coupled LSI-11s each connected by a packet-switching front end to two of many high-speed busses. MICROS allows many users to each run multicomputer programs controlled by UNIX-like commands. MICROS consists of both local and global system modules. The same local modules are resident in each node to load task code and to pass messages. Global operating system tasks are dynamically loaded into selected nodes and cooperate to manage network resources in successively more global nested subtrees. MICROS will eventually include initialization routines to select a virtual tree of resource management nodes within arbitrarily connected networks of thousands of nodes. A new version of MICROS with tools for developing and debugging large distributed application programs is being coded in MODULA-2.

[Zhongxiu 83] Zhongxiu, S.; Du, Z.; Peigen, Y.
ZCZOS: A Distributed Operating System for a LSI-11 Microcomputer Network.
Operating Systems Review 17(3):30-34, July, 1983.

Abstract

Presents ZOZOS, the operating system for the ZOZ distributed microcomputer system. The system may be constructed by any number of LSI-11 microcomputers in any structure, although for the time being the authors have only 5 machines connected in a tree structure. They have designed the ZOZ system for investigating distributed programming as well as for teaching. It is hoped that the system may work as a multiuser time-sharing system with the advantages of extensibility and robustness.

A.2. Interprocess Communication

[Akkoyunlu 72] Akkoyunlu, Erap A., Arthur J. Bernstein and Richard E. Schantz.
An Operating System for a Network Environment.
In *Proceedings, Symposium on Computer-Communications Networks and Teletraffic*, pages 529-538. Polytechnic Institute of Brooklyn, April, 1972.

Abstract
The design of an operating system for a network environment is given. Processes in the system utilize the same set of primitives for communicating with files, devices, or other processes. This permits uniform access to files regardless of their physical location in the network. This system has a modular structure similar to that developed by Dijkstra[1,2].

[Akkoyunlu 74] Akkoyunlu, Erap A., Arthur J. Bernstein and Richard E. Schantz.
Interprocess Communication Facilities for Network Operating Systems.
Computer 7(6):46-55, June, 1974.

Abstract
The connection of several computers into a network poses new problems for the operating system designer. In order to appreciate these problems fully, it is useful to look briefly at networks from the point of view of their goals, their possible configurations, and their level of integration.

The term "computer network" refers not only to the hardware connection between several computers, but also to the software mechanisms for orderly interaction between these machines. This communication facility is the crucial factor in networks. Typical objectives in connecting computers into a network are load sharing, hardware resource sharing, and software resource sharing.

[Akkoyunlu 75] Akkoyunlu, Erap A.
On the Limitations of Acknowledgment Messages.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 37-39. ACM, March, 1975.

Abstract
An important decision, made early in the design of an interprocess communication (IPC) facility, is the amount of information the system undertakes to provide the sender of a message on the final disposition of it. From the point of view of the user, the sender should ideally be supplied with enough status information to allow him to distinguish at least between the following possibilities,

1. the message reached its destination,
2. the intended receiver is not currently in the system,
3. there was a transmission error,
4. the message got timed out (either the destination process itself or the transmission channel was too busy to handle the message with a specific time limit),

since each of these alternatives would suggest a different course of action,

1. go on,
2. give up,
3. try again,
4. right away, re-transmit, perhaps later - meanwhile do something else.

If the system being designed has a high degree of centralized control (as when the appearance of parallel processing is created by multiplexing a single processor), this type of support is fairly easy to provide with very little loss in the elegance of the design, so that there is no problem.

[Ball 76] Ball, J. Eugene, Jerome Feldman, James R. Low, Richard Rashid and Paul Rovner.
RIG, Rochester's Intelligent Gateway: System Overview.
IEEE Transaction on Software Engineering SE-2(4):321-328, December, 1976.

Abstract
Rochester's Intelligent Gateway (RIG) system provides convenient access to a wide range of computing facilities. The system includes five large minicomputers in a very fast internal network, disk and tape storage, a printer/plotter and a number of display terminals. These are connected to larger campus machines (IBM 360/65

and DEC KL10) and to the ARPANET. The operating system and other software support for such a system present some interesting design problems. This paper contains a high-level technical discussion of the software designs, many of which will be treated in more detail in subsequent reports.

[Ball 79a] Ball, J. Eugene, Edward J. Burke, Ilya Gertner, Keith A. Lantz and Richard F. Rashid.
Perspectives on Message-Based Distributed Computing.
In *Proceedings, Computer Networking Symposium*, pages 46-51. IEEE, 1979.

Abstract

At the University of Rochester we have had five years of experience in the design and implementation of a multiple machine, multiple network system called RIG. The design of RIG is based on a model of distributed computation -- independent processes communicating only by messages -- which allows programmers to ignore the details of network and system configuration. This paper describes those aspects of the RIG design which make this isolation from network realities possible. In addition, we describe the styles of message communication which have evolved in RIG.

[Ball 79b] Ball, J. Eugene, J. R. Low and G. J. Williams.
Preliminary ZENO Language Description.
ACM - SIGPLAN Notices 14(9):17-34, September, 1979.

Abstract

The specification of ZENO, a programming language intended as the target language for a research project in advanced compiling, is presented. The language is strongly based on EUCLID, with modifications for message-based parallel processing and a somewhat different treatment of data types.

[Balzer 71] Balzer, R. M.
PORTS -- A Method for Dynamic Interprogram Communication and Job Control.
In *Proceedings, National Computer Conference*, pages 485-489. AFIPS, May, 1971.

Abstract

Without communication mechanisms, a program is useless. It can neither obtain data for processing nor make its results available. Thus every programming language has contained communication mechanisms. These mechanisms have traditionally been separated into five categories based on the entity with which communication is established. The five entities with which programs can communicate are physical devices (such as printers, card readers, etc.), terminals (although a physical device, they have usually been treated separately), files, other programs, and the monitor. Corresponding to each of these categories are one or more communication mechanisms, some of which may be shared with other categories.

[Banino 80] Banino, Jean-Serge, Alain Caristan, Marc Guillemont, Gerard Morisset and Hubert Zimmermann.
Chorus: An Architecture for Distributed Systems.
Technical Report 42, Institut National de Recherche en Informatique et en Automatique (INRIA), November, 1980.

Abstract

The CHORUS project deals with distributed systems; more precisely, it investigates the impact of distribution on operating systems and on execution of applications. This report is the result of the first step in this work. It presents successively:

- a synthesis of the main advantages and constraints of distribution,
- a model for the execution of a distributed application, where communication, synchronization, control, etc... is based on the exchange of messages,
- a model for the construction of a distributed application which permits to turn distribution to the best account,
- examples which illustrate various aspects of the architecture.

This report presents also the minimal functions required from a kernel of operating system in order to support execution of such distributed applications.

[Barter 78] Barter, C. J.
Communications Between Sequential Processes.
Technical Report 34, Department of Computer Science, University of Rochester, November, 1978.

Abstract

In this paper we consider programs which are designed and specified as systems of sequential processes, communicating with each other explicitly, by passing messages. Of central importance in such systems is the way in which communication paths or connections are specified: we particularly wish to point to the works of Feldman (77), Hoare (77) and Milne and Milner (77), by way of contrast with each other and with the present work. We wish to make two specific proposals conerning the specification of inter-process communication: the first defines the "construction" of a message as the determining attribute of message passing, the second gives a communications significance to the structure of hierarchies of processes. We present these proposals within the framework of a small language.

1. Message passing is the sole means of inter-process communication, thereby excluding communication via common data or global variables.
2. For the specification of sequential processes, we adopt the guarded command notation of Dijkstra (75), together with Hoare's (77) extension of that notation to include the possiblitity of an "input command" as part of a guard. This extension greatly enhances the guarded command notation in a multi-process situation (see later).
3. We assume asynchronous, buffered communication, and a few convenient operations which allow messages to be treated as record-like data objects (Feldman (77)).

[Baskett 77] Baskett, Forest, John H. Howard and John T. Montague.
Task Communication in DEMOS.
In *Proceedings, Sixth Symposium on Operating Systems Principles*, pages 23-31. ACM, November, 1977.

Abstract

This paper describes the fundamentals and some of the details of task communication in DEMOS, the operating system for the CRAY-1 computer being developed at the Los Alamos Scientific Laboratory. The communication mechanism is a message system with several novel features. Messages are sent from one task to another over links. Links are the primary protected objects in the system; they provide both message paths and optional data sharing between tasks. They can be used to represent other objects with capability-like access controls. Links point to the tasks that created them. A task that creates a link determines its contents and possibly restricts its use. A link may be passed from one task to another along with a message sent over some other link subject to the restrictions imposed by the creator of the link being passed. The link based message and data sharing system is an attractive alternative to the semaphore or monitor type of shared variable based operating system on machines with only very simple memory protection mechanisms or on machines connected together in a network.

[Bernstein 75] Bernstein, Arthur J. and K. Ekanadham.
Inter-Process Communication in a Network.
Infotech State of the Art Report(24):415-435, 1975.

Abstract

The recent trend in operating system development has been increasingly towards large and complex systems. The introduction of computer networks has only served to compound the problem. Unfortunately, this complexity has brought with it a number of serious problems. The cost of building such systems is enormous. Development time is long and unpredictable, system modification is difficult and the software is never completely debugged.

In order to overcome these difficulties some systems have been constructed in a modular fashion. The code to perform a particular function is localized to a single module and functions are chosen so that a minimum amount of information must be passed across module boundaries. Strict conventions are established concerning the procedure for entering the module and the mechanism for passing information between modules. This approach parallels techniques that have been used for many years in the development of computer hardware.

[Boebert 78a] Boebert, W. Earl.
The HXDP Executive Interim Report.
Technical Report 78SRC53, Honeywell Systems & Research Center, June, 1978.

Abstract

This interim report presents the results of the first phase of the HXDP executive project.

The activities of this phase were primarily conceptual and speculative. They resulted in the concepts and facilities of the executive, as well as a collection of conclusions and observations on the nature of software in the HXDP environment. The report gives the background of the HXDP executive project, describes some of the management and procedural practices used, and lists the lessons learned about research or advanced development projects in the software area. The report presents the resulting concepts and facilities, followed by a rationale. This section traces, as completely as the project team can recall, the influence of the various objectives and constraints on the final concepts and facilities of the executive; it will be the principal section of interest to students of the design process.

The report concludes with some general observations on the nature of software for distributed systems and areas for future research.

[Boebert 78b] Boebert, W. Earl.
Concepts and Facilities of the HXDP Executive.
Technical Report 78SRC21, Honeywell Systems & Research Center, March, 1978.

Abstract

This document presents the Concepts and Facilities of the HXDP Executive. The term "Concept" in this document refers to the abstractions an application programmer uses to visualize his application, describe it to other programmers, and discuss options of design; "Facilities" are the external manifestations of the Executive mechanisms which implement the concepts. The document therefore explicitly presents the two aspects of any general purpose system: the functions it provides and the viewpoint which is imposed or encouraged by their use.

[Boebert 78c] Boebert, W. Earl, William R. Franta, E. Douglas Jensen and Richard Y. Kain.
Decentralized Executive Control in Distributed Computer Systems.
In *Proceedings, COMPSAC 78*, pages 254-258. IEEE, November, 1978.

Abstract

This paper discusses the issues involved in building a real-time control system using a message-directed distributed architecture. We begin with a discussion of the nature of real-time software, including the viability of using hierarchical models to organize the software. Next we discuss some realistic design objectives for a distributed real-time system including fault isolation, independent module verification, context independence, decentralized control and partitioned system state. We conclude with some observations concerning the general nature of distributed system software.

[Boebert 78d] Boebert, W. Earl, William R. Franta, E. Douglas Jensen and Richard Y. Kain.
Kernel Primitives of the HXDP Executive.
In *Proceedings, COMPSAC 78*, pages 595-600. IEEE, November, 1978.

Abstract

This paper describes the kernel of an Executive being implemented for the Honeywell Experimental Distributed Processor (HXDP) -- a vehicle for research in distributed computers for real-time control. The kernel provides message transmission primitives for use by application programs or higher level executive functions. In the paper we describe the message transmission primitives provided by the the kernel and the rationale for their selection based upon the objectives and constraints described in a companion paper.

[Boebert 80] Boebert, W. Earl, Dennis T. Cornhill, William R. Franta, E. Douglas Jensen and Richard Y. Kain.
Communications in the HXDP Executive: Design Issues and Kernel Primitives.
[possibly unpublished].

Abstract

This paper describes the kernel of an Executive for the Honeywell Experimental Distributed Processor (HXDP) -- a vehicle for research in distributed computers for real-time control. The kernel provides message transmission primitives for use by application programs or higher level executive functions.

We begin with a discussion of the nature of real-time software, including the viability of using hierarchical models to organize the software. Next we discuss some realistic design objectives for a distributed real-time system including fault isolaton, independent module verification, context independence, decentralized control and partitioned system state. We describe the message transmission primitives provided by the kernel and the rationale for their selection based upon the objectives and constraints. We conclude with some observations concerning the general nature of distributed system software.

[Boebert ??] Boebert, W. Earl, William R. Franta, E. Douglas Jensen and Richard Y. Kain.
The HXDP Executive: Design Issues and Kernel Primitives.
[possibly unpublished].

Abstract

This paper describes the kernel of an Executive being implemented for the Honeywell Experimental Distributed Processor (HXDP) -- a vehicle for research in distributed computers for real-time control. The kernel provides message transmission primitives for use by application programs or higher level executive functions. In this paper we describe some objectives and constraints of real-time control systems, the message transmission primitives provided by the kernel, and the rationale for this kernel design, based upon the objectives and constraints. We conclude with some general observations on the nature of distributed software.

[Bos 81] Bos, Jan van den, Rinus Plasmeijer and Jan Stroet.
Process Communication Based on Input Specifications.
ACM Transactions on Programming Languages and Systems 3(3):224-250, July, 1981.

Abstract

Input tools, originally introduced as a language model for interactive systems and based on high-level, input-driven objects, have been developed into a model for communicating parallel processes, called the input tool process model (ITP). In this model every process contains an input rule, comparable to the right-hand side of a production rule. This rule specifies in an expression the patterns and sources of input it expects and where the input is to be handled. The reception of the input triggers action inside the tool process. As part of the action, messages may be sent to other processes, with destination specified to a varying degree of identification. A potential candidate for a message is any tool process with the correct type of message slot. Because sending tool processes do not have to specify completely the identity of receiving tool processes, and vice versa, ITP provides a fully dynamic communication model. Most communication aspects of other recently developed models are contained in this model. Synchronization of processes is accomplished implicitly by the input specification; explicit synchronization constructs such as monitors and guarded regions can therefore be easily simulated. The ITP constructs provide a general concept for interprocess communication. Its application areas range from interaction via process control to operating systems. From a programming point of view, the language constructs offered are not in any way dependent on whether processes run on single or multiple processors.

[Brinch Hansen 77]
Brinch Hansen, Per.
Network: A Multiprocessor Program.
In *Proceedings, Computer Software & Applications Conference*, IEEE, November, 1977.
[pages].

Abstract

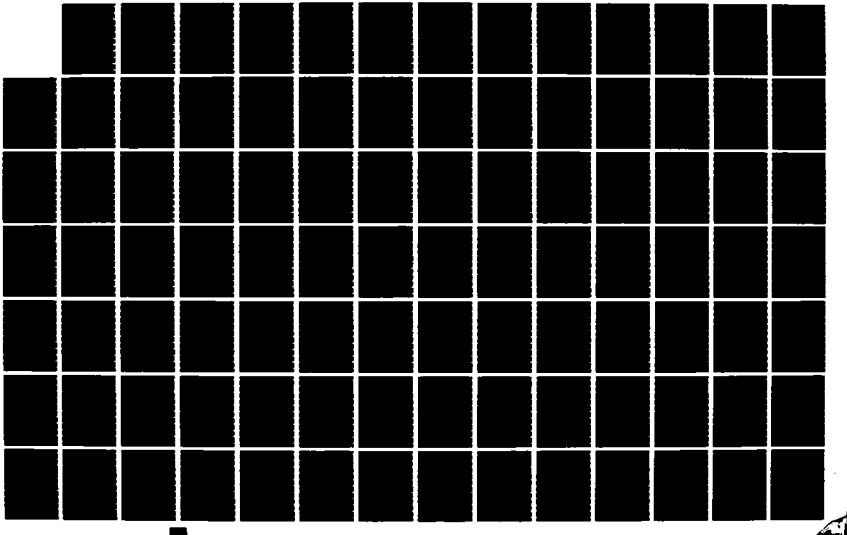
This paper explores the problems of implementing arbitrary forms of process communication on a multiprocessor network. It develops a Concurrent Pascal program that enables distributed processes to communicate on virtual channels. The cannels cannot deadlock and will deliver all messages within a finite time. The operation, structure, text, and performance of this program are described. It was written, tested, and described in 2 weeks and worked immediately.

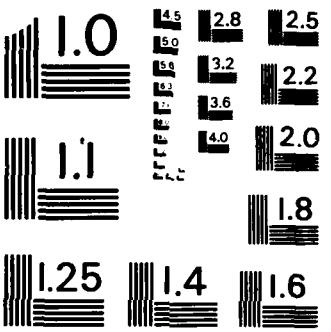
[Brinch Hansen 78]
Brinch Hansen, Per.
Distributed Processes: A Concurrent Programming Concept.
Communications of the ACM 21(11):934-942, November, 1978.

Abstract

A language concept for concurrent processes without common variables is introduced. These processes communicate and synchronize by means of procedure calls and guarded regions. This concept is proposed for real-time applications controlled by microcomputer networks with distributed storage. The paper gives several examples of distributed processes and shows how they include procedures, coroutines, classes, monitors, processes, semaphores, buffers, path expressions, and input/output as special cases.

AD-A169 754 DECENTRALIZED SYSTEM CONTROL(U) CARNEGIE-MELLON UNIV 3/4
PITTSBURGH PA DEPT OF COMPUTER SCIENCE
E D JENSEN ET AL. APR 86 CMU-CS-ARCHONS-83-1
UNCLASSIFIED RADC-TR-85-199 F30602-81-C-0297 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS - 1963 - A

[Britton 80] Britton, Dianne E. and Mark E. Stickel.
An Interprocess Communication Facility for Distributed Applications.
In *Digest of Papers, COMPCON 80 Fall*, pages 590-595. IEEE, 1980.

Abstract

When an application is distributed across several processor nodes, the facilities available for communication and synchronization have a tremendous influence on the ease with which the application program can be designed, written, and understood. This paper presents a framework for structuring a distributed application as a set of concurrent processes and describes a message-based interprocess communication and synchronization facility. This facility, which is supported in a prototype implementation by a kernel-executive called SUPPOSE, is particularly appropriate for loosely-coupled networks where common memory cannot be assumed.

[Cashin 80] Cashin, Peter M.
Inter-Process Communication.
Technical Report 8005014, Bell-Northern Research, June, 1980.

Abstract

This report gives a survey of both procedure oriented and message oriented inter process communication techniques. It compares these techniques and discusses their use in distributed systems. The report forms the basis for lectures to be given at the NATO Advanced Study Institute on Multiple Processors, Maratea, Italy, June 1980.

The purpose of this report is to survey and compare many of the different schemes used for inter process communication, and to draw out the key issues for inter process communications in distributed systems. It should become clear from this survey that inter process communications are far from being well understood, several significant steps have occurred over the last few years but we are still some way from a wide spread concensus and a well proven set of tools.

[Cheriton 79] Cheriton, David R., Michael A. Malcolm, Lawrence S. Melen and Gary R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, February, 1979.

Abstract

Thoth is a real-time operating system which is designed to be portable over a large set of machines. It is currently running on two minicomputers with quite different architectures. Both the system and application programs which use it are written in a high-level language. Because the system is implemented by the same software on different hardware, it has the same interface to user programs. Hence, application programs which use Thoth are highly protable. Thoth encourages structuring programs as networks of communication processes by providing efficient interprocess communication primitives.

[Cheriton 80] Cheriton, David R.
A Loosely Coupled I/O System for a Distributed Environment.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 297-318. IBM, August, 1980.

Abstract

The design of a loosely coupled I/O system is presented that provides a byte-oriented and block-oriented I/O abstraction in a distributed environment. The design is based on a simple protocol between client processes and I/O server processes. The I/O system is loosely coupled in the sense that it exists as a protocol or convention among the client processes and the server processes.

The I/O system consists of: a library of functions that implements the protocol in terms of a set of message primitives, a set of participating I/O server processes, and an I/O server and file identification scheme that supports symbolic naming of files. The function library makes this underlying structure transparent to the application programmer. The message primitives make the protocol implementation independent of the underlying network configuration and hardware.

[Chesley 81] Chesley, Harry R. and Bruce V. Hunt.
Squire - A Communications-Oriented Operating System.
Computer Networks 5(2):333-339, 1981.

Abstract

This paper presents the architecture of a communication-oriented, real-time operating system named Squire. The Squire kernel provides memory management, preemptive multitaking, interprocess communication, and the ability

to manage data outside the process address space, as well as services such as timers. User processes are protected from one another by means of restrictions on what objects they can access and on the type of access. Squire has been designed to provide efficient communication between cooperating processes, portability to new machine architectures, and support for multiple processor and distributed processor usage. Protection, reliability, and robustness have been major design goals. Squire supports a new kind of object called chunks, which exist outside the process address space, and can be used to store and manage data. Squire also supports a means for extending the kernel in a controlled manner; this mechanism is used both to implement such traditional functions as device drivers and to provide extended kernel services not present in the basic Squire kernel.

[Chessen 80] Chesson, G. L. and A. G. Fraser.
Datakit Software Architecture.
In *Digest of Papers, COMPCON 80 Spring*, pages 59-61. IEEE, 1980.

Abstract

Datakit packet switching and data transmission modules provide a local area networking capability for a range of applications and traffic types. The extent to which communication facilities of this kind can be utilized, extended, and maintained strongly depends on the nature of the related software environment. The software evolving with Datakit represents a step toward a set of general-purpose software building blocks that can be used with different communication hardware, different computers, and, to some degree, with different operating systems.

[Clark 82a] Clark, David. D.
Name, Addresses, Ports, and Routes.
[RFC814].

Abstract

It has been said that the principal function of an operating system is to define a number of different names for the same object, so that it can busy itself keeping track of the relationship between all of the different names. Network protocols seem to have somewhat the same characteristics. In TCP/IP, there are several ways of referring to things. At the human visible interface, there are character string "names" to identify networks, hosts, and services. Host names are translated into network "addresses", 32-bit values that identify the network to which a host is attached, and the location of the host on that net. Service names are translated into a "port identifier", which in TCP is a 16-bit value. Finally, addresses are translated into "routes" which are the sequence of steps a packet must take to reach the specified addresses. Routes show up explicitly in the form of the internet routing options, and also implicitly in the address to route translation tables which all hosts and gateways maintain.

This RFC gives suggestions and guidance for the design of the tables and algorithms necessary to keep track of these various sorts of identifiers inside a host implementation of TCP/IP.

[Clark 82b] Clark, David D.
Modularity and Efficiency in Protocol Implementation.
[RFC817].

Abstract

Many protocol implementers have made the unpleasant discovery that their packages do not run quite as fast as they had hoped. The blame for this widely observed problem has been attributed to a variety of causes, ranging from details in the design of the protocol to the underlying structure of the host operating system. This RFC will discuss some of the commonly encountered reasons why protocol implementations seem to run slowly.

Experience suggests that one of the most important factors in determining the performance of an implementation is the manner in which that implementation is modularized and intergrated into the host operating system. For this reason, it is useful to discuss the question of how an implementation is structured at the same time that we consider how it will perform. In fact, this RFC will argue that modularity is one of the chief villains in attempting to obtain good performance, so that the designer is faced with a delicate and inevitable tradeoff between good structure and good performance. Further, the single factor which most strongly determines how well this conflict can be resolved is not the protocol but the operating system.

[Collier 72] Collier, W. W. and P. H. Gum.
Wait-Free Interprocess Communication Mechanisms.
IBM Systems Journal 14(12), May, 1972.

Abstract

The following series of programmable routines allow one specific process (i.e., program) in a computer system to send an indefinite number of messages to exactly one other process. No message may be lost or received out of order. Once the sender has completed sending a message, the receiver must be able to receive the message (this

rules out the case in which the sender tries, but fails, to send a message and so it tries again when it next sends another message). Each process must operate in such a fashion that the other process cannot tell that the first process is active. In particular, neither process can wait for the other to become inactive, such as in a multiprogrammed computer system.

[Cook 80] Cook, Robert P.
The Starmod Distributed Programming System.
In *Digest of Papers, COMPCON 80 Fall*, pages 729-735. IEEE, 1980.

Abstract

Distributed programming is characterized by high communication costs and the absence of shared variables and procedures as synchronization tools. StarMod is a language, derived from Modula, which is intended for systems programming in the network environment. The StarMod system attempts to address the problem areas in distributed programming by creating an environment which is conducive to efficient and reliable network software construction. The StarMod system will include program packages for compilation, debugging, and software maintenance as well as for performance evaluation and modeling.

[Cornhill 79] Cornhill, Dennis T. and W. Earl Boebert.
Implementation of the HXDP Executive.
[CH1939].

Abstract

This paper describes a first implementation of the executive for the Honeywell Experimental Distributed Processor (HXDP). HXDP has been built to investigate distributed, decentralized control in real time applications.The purpose of the implementation is to demonstrate the utility of, and to gain experience with the executive primitives in the area of interprocess communication.

[Cox ??] Cox, George W., William M. Corwin, Konrad K. Lail and Fred J. Pollack.
Interprocess Communication and Processor Dispatching on the Intel 432.
[submitted for publication, 1982].

Abstract

This paper describes a unified facility for interprocess communication and processor dispatching on the Intel 432. The facility is based on a queuing and binding mechanism called a port. The paper describes our goals and motivations for ports, both abstract and implementation view of ports and their absolute and comparative performance.

[Dallas 80] Dallas, I. N.
A Cambridge Ring Local Area Network Realisation of a Transport Service.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 271-296. IBM, August, 1980.

Abstract

A Network Independent Transport Service has been defined in the United Kingdom. From the service description, various protocols can be derived to provide the service over particular communications media. The paper gives a brief description of this Transport Service and goes on to describe its realisation, (encoding), for the Cambridge Ring Local Area Network in operation at the University of Kent. This realisation uses an existing Ring protocol. The conclusions derived from the project are given at the end of the paper.

[Dannenberg 81] Dannenberg, Roger B.
AMPL: Design, Implementation, and Evaluation of a Multiprocessor Language.
Technical Report ?, Computer Science Department, Carnegie-Mellon University, March, 1981.

Abstract

AMPL is an experimental high-level language for expressing parallel algorithms which involve many interdependent and cooperating tasks. AMPL is a strongly-typed language in which all inter-process communication takes place via message passing. The language has been implemented on the CM* multiprocessor, and a number of programs have been written to perform numeric and symbolic computation. In this report, the design decisions relating to process communication primitives are discussed, and AMPL is compared to several other languages for parallel processing. The implementation of message passing, process creation, and parallel garbage collection are described. Measurements of several AMPL programs are used to study the effects of language design decisions upon program performance and algorithm design.

[Danthine 75] Danthine, Andre' A. S. and Joseph Bremer.
Communication Protocols in a Network Context.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, ACM, March, 1975.

Abstract

If the problem of process cooperation has been intensively studied since 1965 [5], it is much more recently [7,12],that the attention has been drawn to the problems associated with the cooperation of distributed processes. In this case, additional problems arise from the communication support and from the environment disparities in terms of space, time and function.

In a distributed computer network based on a packet-switched communication subnet, it is possible to describe the system as a hierarchical structure. We will consider here only three levels:

- Level 0 : communication between nodes of the subnets.
- Level 1 : communication between hosts connected to the net.
- Level 2 : communication between users processes (subscribers) in different hosts.

At each level, the communication is based on a protocol and the data structures to be exchanged are different. For instance the data structure exchanged at level 2 may be a sequential file. As there is no direct support of communication at this level, it is through the mechanisms of level 1 and 0 that the transfer will take place with data structure at each level not directly related to the upper level one. The complete definition of the system will therefore required not only the level 0, 1 and 2 protocols, but also inter-level protocols.

In the following, we will concentrate on the level 1 protocol. It is the basic communication protocol of a network since it will be used by user processes in different hosts and it will use the subnet as a communication support. This level 1 entity is called a "TS" (transport station) in CYCLADES [13] and a "TCP" (transmission control program) in [2].

[Danthine 80] Danthine, Andre' A. S. and F. Magnee.
Transport Layer - Long-Haul Vs. Local Network.
In *Proceedings, IFIP Working Group 6.4, International Workshop on Local Networks*, pages 271-296. IBM, August, 1980.

Abstract

A computer network may be considered as a set of cooperating ditributes processes organized in a hierarchical structure.

[Danthine 81] Danthine, Andre' A. S.
Design Principles of Communication Protocols.
In *Data Communication and Computer Networks*, pages 257-273. IFIP, 1981.

Abstract

The central concept in a hierarchical model of a computer network is the transport service. Besides providing the processes with network wide name space it allows a connection oriented communication. The designer choices are related to the method to construct the network wide name space, the data elements on which is based the connection oriented communication, the existence of interrupt facilities and of lettergram communication.

The process/transport interface allows the access not only to the network wide service but also to the additional services which may be offered by a station to its local processes.

The transport protocol is responsible for achieving the service offered and is based on the expected performances of the transmission service. The designer choices are related to the connection opening scheme and the data elements on which is based on the error control and the flow control.

[desJardings 75] desJardings, Richard.
Semantic Notions for Interprocess Communication.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 159-162. ACM, March, 1975.

Abstract

It is proposed that a process, operating within a virtual address space (VAS), always communicate with all processes outside its VAS by a single uniform mechanism. This mechanism, which may be implicit rather than explicit in a procesor with virtual addressing hardware, is a set of generalized I/O primatives, as suggested in

some detail by [Akkoyynlu, Bernstein, and Schantz]. Such a primitive designates a segment (buffer) to be realized in the VAS of a designated receiving process. We consider the problems of designating the receiver process in the sequel, and for now concentrate on the structure of the sender.

[Didic 82] Didic, Milena and Bernd Wolfinger.
Simulation of a Local Computer Network Architecture Applying a Unified Modeling System.
Computer Networks 6(1):75-91, 1982.

Abstract

A modeling system is described which allows us to determine qualitative and quantitative interdependences between system parameters for protocol hierarchies and for various organizations of services in computer networks. Its use is shown for modeling a local resource sharing network specified in terms of the ISO-Reference Model of Open Systems Interconnection. The simulation is intended to help both during the design phase of the network architecture in order to find an optimum design solution and subsequently, after implementation, to investigate trade-offs among various network configurations. Experimental results to increase the efficiency of a network configuration are given.

[Ekanadham 75] Ekanadham, K. and Arthur J. Bernstein.
The Structure of Interprocess Communication.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 28-30. ACM, March, 1975.

Abstract

The purpose of this work is to make some general observations about the structure of any Interprocess Communication mechanism (IPCM). Much of the work in this area, to date, has confined itself to the design of a specific IPCM to meet the needs of a particular operating system environment. It is our feeling that there are certain basic principles which underlie the structure of any IPCM. An understanding of these principles should help to clarify various tradeoffs and shed some light on the design process.

[Elovitz 74] Elovitz, Honey S. and Constance L. Heitmeyer.
What is a Computer Network?
In *IEEE 1974 NTC Record*, pages 1007-1014. IEEE, 1974.

Abstract

A recent trend in computer systems has been the use of data transmission and packet switching technology to construct what are commonly referred to as "computer networks". There is an important, yet often unmentioned, distinction among such networks, namely between "computer communications networks" and "computer networks". In a "computer-communications network", the user must explicitly manage the computer resources. In a "computer network", these resources are managed automatically by a network operation system. Most of the existing "computer networks" such as TYMNET and ARPANET are more accurately labeled "computer-communications networks".

This paper intends to remove the obscurity from the term "computer network" by characterizing the differences between "computer networks" and "computer-communications networks". Several existing networks are described and classified.

[Enslow 79] Enslow Jr., Philip H. and Robert L. Gordon.
Interprocess Communication in Highly Distributed Systems -- A Workshop Report.
Technical Report GIT-ICS-79/09, Georgia Institute of Technology, December, 1979.

Abstract

The subject of the workshop is Interprocess Communication Mechanisms with a particular focus on process to process communications in highly distributed systems. Highly distributed systems are characterized by a very high degree of loose-coupling between physical resources as well as between logical resources plus dynamic, short-term changes in the topology and organization of the total system. These characteristics place new requirements on the design and performance of IPC mechanisms that are assuming extreme importance in advancing the state-of-the-art in all forms of distributed systems.

[Farber 72a] Farber, David J. and Kenneth C. Larson.
The System Architecture of the Distributed Computer System--The Communications System.
In *Proceedings, Symposium on Computer-Communications Networks and Teletraffic*, pages 21-27. Polytechnic Institute of Brooklyn, April, 1972.

Abstract

The Distributed Computing System (DCS) is an experimental computer network under study at the University of California at Irvine under NSF funding. The network has been designed with the following goals in mind: reliability, low cost factilities, easy addition of new processing services, modest startup cost, and low incremental expansion cost. The structure chosen to achieve these goals is a digital communications ring using T1 technology and fixed message lengths. The computers used are small to medium scale and are interfaced to the ring using a fairly sophisticated piece of hardware called a Ring Interface (RI).

There are two features which make the communications protocols unique. First, messages are addressed to processes, not processors. This is accomplished by placing an associative store in each RI. The store contains the names of all processes active on the attached processor. When a message arrives over the ring, the destination process name is matched against the associative store. If a match occurs the message is copied and passed over the ring to the next RI. Second, messages are only removed at the RI from which they originate. The ring may be thought of as a series of message slots. To transmit a message the RI waits for an empty slot and places the message on the ring. The message is copied when necessary as it is removed from the ring. If errors are detected or the message fails to return in a specific amount of time the message is retransmitted. The retransmission causes problems since RIs may receive multiple copies of the message. The paper describes a scheme for sequencing messages which removes these problems. Note that this scheme allows messages top be broadcast to all processes or a class of processes. The DCS/OS software uses this feature extensively. The paper also discusses the error detection and maintenance features. Basically, each RI has a *short circuit* which removes it from the ring which maintaining the ring connectivity. This short circuit can be activated through internal checks within the RI or externally by specific messages. Redundancy of communication paths in the ring protects the ring connectivity.

[Farber 72b] Farber, David J. and Kenneth C. Larson.
The Structure of a Distributed Computing System-Software.
In *Proceedings, Symposium on Computer-Communications Network and Teletraffic*, pages 539-544. Polytechnic Institute of Brooklyn, April, 1972.

Abstract

This paper describes a software system which allows the control of a network of small processors to be distributed among the processors on the network. The design goals for the software system are presented, the primary goal being that the network be fail-soft (Section V). The hardware used to implement the network is then described. The unique feature of the hardware is a technique of message addressing which allows processes to communicate with no knowledge of each other's physical location in the network. The next section shows the ways in which the operating system was shaped by the design goals and describes the interprocess communications scheme and some of the basic characteristics of the operating system. A more detailed description of the entire operating system is then presented, in particular showing the ways in which the responsibility for resouce allocation and scheduling is distributed among the separate processors. The software which maintains the network is described and examples of error conditions and recovery or checking procedures are given. The future plans for the network are presented.

[Farber 76] Farber, J. and R. Pickens.
The Overseer, a Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures.
In *Proceedings, Third International Conference on Computer Communication*, pages 441-451. August, 1976.

Abstract

Thin wire communications, otherwise known as serial message sending, encourages modularity in distributed program design and makes visible the interprocess communications streams to an unprecedented degree. In this paper, a powerful process monitoring capability, the overseer function, is proposed to aid the program developer in guaranteeing the dynamic correctness of his distributed process mix. The top down design process is overviewed with the emphasis on generating an analyzable model of the intra-module control structure. With appropriate augmentation of interprocess communications streams it is feasible to endow the communications

with a control sequence validation capability. The need for dynamic changing process contexts is discussed, and the overseer is shown to be capable of emulating this level of process behavior. Path verification (for protection) and single channel monitoring (for dynamic probing) are two final attributes which may usefully be part of the overser function. Overall the overseer is only a part of a systematized process for distributed system design, but promises great potential in improving the visibility of dynamic process behavior in distributed systems.

[Feldman 79] Feldman, Jerome A.
High Level Programming for Distributed Computing.
Communications of the ACM 22(6):353-368, June, 1979.

Abstract

Programming for distributed and other loosely coupled systems is a problem of growing interest. This paper describes an approach to distributed computing at the level of general purpose programming languages. Based on primitive notions of module, message, and transaction key, the methodology is shown to be independent of particular languages and machines. It appears to be useful for programming a wide range of tasks. This part of an ambitious program of development in advanced programming languages, and relations with other aspects of the project are also discussed.

[Fjellheim 79] Fjellheim, Roar A.
A Message Distribution Technique and its Application to Network Control.
Software-Practice and Experience 9(?):499-505, June, 1979.

Abstract

The patterns of message exchange in distributed computer systems can become sufficiently complex to justify the construction of communication services that extend the basic message transmission mechanism. A simple method for implementing a copy distribution, or broadcast, service is described. It is shown how the method can support command and monitoring functions in a computer communication network.

[Fleisch 81] Fleisch, Brett D.
An Architecture for Pup Services on a Distributed Operating System.
SIGOPS-Operating Systems Review 15(1):26-44, January, 1981.

Abstract

At the University of Rochester the computer science department has had six years of experience in the design and implementation of a multiple-machine, multiple network distributed system called RIG. Rochester's Intelligent Gateway (RIG) [1,2,3] is a dual processor gateway which connects three computer networks to provide convenient access to a wide range of computer facilities. RIG was built to serve as an intermediary between the human user (working through a display terminal or personal computer) and a variety of computer systems. The bulk of the user's computational requirements is met by these systems, which are either partially integrated into the RIG system through a fast local network or loosely coupled to it through the ARPANET. RIG also provides a number of basic services such as printing, plotting, local file storage, and support for a number of display terminals.

This paper presents an architecture for Pup services on RIG. Pup is the name of an internetwork packet format (PARC Universal Packet), a hierarchy of protocols and a style of internetwork communication [4]. These services proposed provide access to Pup interprocess communication primitives on a distributed operating system. The motivation for this design is twofold. First, we wish to develop a framework in which processes may perform network communication using a wide variety of interprocess communication styles, selectable by the process upon initialization. These styles are necessary because of the diversity of protocols in the environment. Moreover, this framework must extend an environment that has provided logical centralization of distributed resources. Second, we wish to integrate some new functions into our message based operating system which are not currently provided. Although many services have been provided by RIG, the provision of Pup services will give us added flexibility.

[Folts 80] Folts, Harold C.
X.25 Transaction-Oriented Features - Datagram and Fast Select.
IEEE Transactions on Communications COM-28(4):496-500, April, 1980.

Abstract

The latest proposed revisions to CCITT Recommendation X.25 for packet-switching service in public data networks now include two new capabilities suitable for transport of small amounts of data. The first provides datagram service for the transport of independent "message type" packets. The other new feature is the fast select facility which provides for the inclusion of 128 octets of user data in the call establishment packets for virtual call service. Both these new provisions greatly enhance the capability of X.25 to efficiently support the broadest

range of user applications.

[Ford 76] Ford, W. S. and V. C. Hamacher.
Hardware Support for Inter-Process Communication and Processor Sharing.
In *Proceedings, Third Annual Symposium on Computer Architecture*, pages 113-118. IEEE, January, 1976.

Abstract

The abstraction of a computer system as a set of asynchronous communicating processes is an important system concept. This paper indicates how the concept could be supported at a low hardware level. A new, inter-process communication mechanism called a mailbox is introduced. Examples of its use as a programming tool are given. This is followed by a description of hardware features that use this mechanism as the basis of communication between the components of a complete system. These features include processor-sharing hardware capable of handling process selection and switching with high efficiency. It is also indicated how these features can take the place of conventional input/output structures.

[Ford 77] Ford, W. S. and V. C. Hamacher.
Low Level Architecture Features for Supporting Process Communication.
The Computer Journal 20(2):156-162, May, 1977.
British Computer Society.

Abstract

A proposal is presented for low level hardware features which would assist in the realisation of the abstraction of a computer system as a set of asynchronous communicating processes. A low level synchronisation and communication mechanism, called a *mailbox*, is described, togeather with details of a hardware structure for configuring a complete system around a set of these mailboxes. Programming for this architecture is then discussed. It is shown how the new features can be used for controlling input/output, and for handling general synchronization.

[Forsdick 81] Forsdick, Harry C., William I. MacGregor, Richard E. Schantz, Steven A. Swernofsky, Robert H. Thomas and Stephen G. Toner.
Distributed Operating System Design Study: Final Report.
Technical Report 4674, Bolt Beranek and Newman Inc., May, 1981.

Abstract

A Distributed Operating System (DOS) is made from many interacting parts. The architecture for a DOS is the organization and relationships between the various components, programs, and protocols that make up the distributed computer system. Specifying a basic architecture for a DOS serves several purposes. It provides an integrated framework to which refinements in the areas of our special concern (Global Resource Control and Reliability) may be made. An explicit architecture records many implications of the goals stated in the previous Chapter which are system-wide implications. Finally, an architectural framework places some boundaries on subsequent aspects of the emerging design.

[Franta 81] Franta, William R., E. Douglas Jensen, Richard Y. Kain and George D. Marshall.
Real-Time Distributed Computer Systems.
Advances in Computers 20:39-82, 1981.

Abstract

Distributed computer systems, containing several computers, may provide increased system availability and reliability. Their design is complex, involving the design of communications mechanisms in hardware and software and the selection of policies and mechanisms for distributed sytem control. The complex design issues may have simple solutions in well-understood application environments; the real-time control environment is one such environment. For these reasons, some early distributed computer system development projects have focused on the real-time application environment.

In this contribution we cover real-time distributed computer systems from promise through design and implementation. First, we discuss the motivation for distributed computer systems in terms of possible system characteristics attained by distributing the computational resources and then we characterize the real-time control application environment. In subsequent sections we review the options and issues related to hardware and software designs for distributed systems, and accompany the general discussions with the details of the design and implementation of the Honeywell Experimental Distributed Computing (Processor) system, known as HXDP. The HXDP project hardware design began in 1974, was realized in 1976, and system software design and realization were completed in 1978. Applications experiments are continuing in 1980.

[Galtieri 80] Galtieri, Cesare A.
Architecture for a Consistent Decentralized System.
Technical Report 36132, IBM, June, 1980.

Abstract

This paper has three principal aims. First, to set forth a definition of system consistency which allows for decentralized systems and which is as free as possible of hidden implementation assumptions. Second, to propose a system architecture which guarantees consistency, in the sense previously defined. Third, to outline an implementation approach for a simple data management function.

The general intent of the proposal is to achieve a high degree of independence and concurrency among the various components of a decentralized system consistency. In particular our approach

- allows for concurrency within a transaction, a capability which is important for the effective support of complex transactions in a decentralized environment;
- guarantees maximal apparent concurrency among transactions;
- facilitates the support of more selective operations which, in most cases, transform apparent concurrency into real concurrency.

[Garlick ??] Garlick, Lawrence L., Raphael Rom and Johathan B. Postel.
Issues in Reliable Host-to-Host Protocols.

Abstract

Fully reliable network host-to-host protocols have recently gained significant attention, primarily due to more stringent security requirements of network users. This paper will discuss issues related to one such protocol, which is supported by the Transmission Control Program (TCP). The protocol, first introduced in 1974, features end-to-end positive acknowledgement, retransmission, internetwork addressing capabilities, and ordered delivery.

The issues of interest in this paper are protocol correctness and completeness, protocol efficiency, and complexity of implementation. The discussion will suggest alterations and extensions to TCP.

Flow control heuristics using TCP's windowing techniques are explored. Flow control information is augmented to allow fair apportionment of bandwidth, better bandwidth utilization through optimistic credits, flow control credits matched to the type of traffic, and increased performance for high precedence connections.

An alternative for selecting the startup sequence number of a connection is presented. It is suggested that the resynchronization method for sequence number space management should be abandoned because it is overly complicated and can actually fail when the data stream is stopped by flow control.

The need for the separation of data and control channels is motivated, introducing the notion of a reliable subchannel.

The findings are presented both to further the understanding of reliable protocols and to encourage intelligent implementations of TCP.

[Gehringer 81] Gehringer, Edward F. and Robert J. Chansler Jr.
StarOS User and System Structure Manual.
Technical Report ?, Department of Computer Science, Carnegie-Mellon University, June, 1981.
[*not released as of Jan. 82*].

Abstract

Technological advances have made it attractive to interconnect many less expensive processors and memories to construct a powerful, cost-effective computer. *Potential* benefits include increased cost-performance resulting from the exploitation of many cheap processors, enhanced reliability in the integrity of data and in the availability of useful processing power, and a physically adaptable computer whose capacity can be expanded or reduced by addition or removal of modular components. Realizing these *potential* benefits requires software structures that make effective use of the hardware. StarOs is a message-based, object-oriented, multiprocessor operating system, specifically designed to support task forces, large collections of concurrently executing processes that cooperate to accomplish a single purpose. StarOS has been implemented at Carnegie- Mellon University on the 50 processor Cm* multimicroprocessor computer.

[Gentleman 80] Gentleman, W. Morven and J. E. Corman.
Design Considerations for a Local Area Network Connecting Diverse Primitive Machines.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 207-221. IBM, August, 1980.

Abstract

Local area networks have typically been designed to connect remote peripherals and concentrators to host machines, or to interconnect homogeneous computers running a common network operating system, or to interconnect substantial self-sufficient computer systems. The objective for the network we are constructing are quite different. Most of the network subscribers will be primitive machines of diverse types. This has implications which strongly affect design decisions in the network hardware and software.

Firstly, it means the subscriber hardware is inexpensive, so to maintain balance, the network cost per subsciber must be low, and, in particular, the hardware interface to the network must be inexpensive too.

Secondly, it means that the machines are of many architectures, so a standard port to the subscriber computer must be used; building custom hardware for each machine type is infeasable.

These two imply the network must present an interface to a standard serial communications port, or to a standard parallel port, if such can be defined.

Third, it means that the operating systems of the subscribers will be quite different, indeed, the same subscriber may, at different times, run several incompatable systems, and one of the requirements of the network is to be able to download such systems. This implies the outer-most level of communications protocol must be very simple, perhaps byte-stream with preset virtual circuits. More flexible protocols must be built on top of this.

Fourth, there is no central machine: groups of subscribers can be expected to communicate heavily among themselves, but rarely with others. Their higher-level protocols should suit them. File transfer will be the main activity.

This paper discusses these and other factors, shows why most existing network designs are inappropriate in this context, then, by describing the network being built at the University of Waterloo, illustrates that suitable designs are possible.

[Giloi 81] Giloi, W. K. and P. Behr.
An IPC Protocol and its Hardware Realization for a High-Speed Distributed Multicomputer System.
In *Proceedings, Eighth Annual Symposium on Computer Architecture*, pages 481-493. IEEE and ACM, 1981.

Abstract

Multicomputer systems with distributed control form an architectue that simultaneously satisfies such design goals as high performance through parallel operation of VLSI processors, modular extensibility, fault tolerance, and system software simplification. The nodes of the system may be locally concentrated or spatially dispersed as a local network. Applications range from data base-oriented transactional systems to "number crunching." The system is service-oriented; that is, it appears to the user as one computer on which parallel processing takes place in the form of cooperating processes. Cooperation is regulated by the unique interprocess communication (IPC) protocol presented in this paper. The high-level protocol is based on the consumer/producer model and satisfies all requirements for such a distributed multicomputer system. It is demonstrated that the protocol lends itself toward a straightforward mechanization by dedicated hardware consisting of a cooperation handler, an address transformation and memory guard unit, and bus connection logic. These special hardware resources, assisted by a "local operating system", form the supervisor of a node. Nodes are connected by a high-speed bus (280 Mbit/sec). Programming aspects as implied by the protocol are also described.

[Green 80] Green Jr., Paul E.
An Introduction to Network Architectures and Protocols.
IEEE Transactions on Communications COM-28(4):413-424, April, 1980.

Abstract

This tutorial paper is intended for the reader who is unfamiliar with computer networks, to prepare him for reading the more detailed technical literature on the subject. The approach here is to start with an ordered list of the functions that any network must provide intieing two end users together, and then to indicate how this leads naturally to layered peer protocols out of which the architecture of a computer network is constructed. After a

discussion of a few block diagrams of private (commercially provided) and public (common carrier) networks, the layer and header structures of SNA and DNA architectures and the X.25 interface are briefly described.

[Guillemont 82] Guillemont, Marc.
The Chorus Distributed Operating System: Design and Implementation.
In *Proceedings, International Symposium on Local Computer Networks*, Institut National de Rechereche en Informatique et en Automatique (INRIA), April, 1982.

Abstract

CHORUS is an architecture for distributed systems. It includes a method for its execution and the (operating) system to support this execution. One important characteristic of CHORUS is that the major part of the system is built with the same architecture as applications. In particular, the exchange of messages, which is the fundamental communication/synchronization mechanism, has been extended to the most basic functions of the system.

[Guillier 80] Guillier, P. and D. Slosberg.
An Architecture with Comprehensive Facilities of Inter-Process Synchronization and Communication.
In *Proceedings, Seventh Annual Symposium on Computer Architecture*, pages 264-270. IEEE and ACM, 1980.

Abstract

In the architecture of the "Level 64" manufactured by CII-Honeywell-Bull and Honeywell Information Systems, processes executing in a central processor are known to the hardware-firmware. They use the same semaphore mechanism as processes executing in an input-output controller. This implies specific data structures recognized by the hardware-firmware and a hardware-firmware dispatching of the central processor resource. Experience in this domain has led to the development of some new extensions.

[Halsall 78] Halsall, F. and A. E. Fenesan.
Software Aspects of a Closely Coupled Multicomputer System.
Computers and Digital Techniques 1(1):21-26, February, 1978.

Abstract

This paper describes the philosophy and structure of the operating-system software which is currently being developed for a closely coupled multicomputer system. The proposed operating system is effectively distributed between the individual computing elements of the system. Each computing element or module contains a copy of a simple operating system or nucleus which has been designed on the one hand to provide a standard software interface for the applications software within the module and on the other to form an interface with other modules through the intercomputer-communication facility. A necessary and sufficient condition for a computing module to function in the proposed system is the possession of a copy of this nucleus. The nucleus software has been implemented in a high-level procedure-based language and is designed to provide the applications programmer with a basic set of commands or primitives which facilitate the creation and control of the other application processes within the same module and the sending and receiving of messages to and from application processes resident within other modules. The paper also includes details of the size and performance of the implemented system.

[Halstead 78] Halstead Jr., Robert H.
Multiple-Processor Implementations of Message-Passing Systems.
PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, January, 1978.

Abstract

The goal of this thesis is to develop a methodology for building networks of small computers capable of the same tasks now performed by single larger computers. Such networks promise to be both easier to scale and more economical in many instances.

The mu calculus, a simple syntactic formalism for representing message-passing computations, is presented and augmented to serve as the semantic basis for programs running on the network. The augmented version includes cells, tokens, and semaphores, allow certain simple communications and synchronization tasks without involving fully general side effects.

The network implementation presented supports object references, keeping track of them by using a new concept,

the reference tree. A reference tree is a group of neighboring processors in the network that share knowledge of a common object. Also discussed are mechanisms for handling side effects on objects and strategy issues involved in allocating computations to processors.

[Hammond 80] Hammond, Richard A.
Experiences with the Series/1 Distributed System.
In *Digest of Papers, COMPCON 80 Fall*, pages 585-589. IEEE, 1980.

Abstract

The Series/1 Distributed System (SODS), developed at the University of Delaware, is an experimental system for research in distributed computing. It consists of several IBM Series/1 computers, a local communications network, an operating system (SODS/OS), a file system (SODS/FS), and applications software. Experience in designing, implementing, and using the system has given insight into the basic strengths and weaknesses of its design.

[Hartenstein ??] Hartenstein, Reiner W., Werner Konrad and Anton Sauer.
A Loosely Coupled Multi-Microstructure as a Tool for Software Development.
[unknown].

Abstract

The communication tools of message-oriented operating systems and related scientific methods and concepts can be directly mapped into hardware constructs. Thus loosely coupled microcomputer networks along with approaches to specification and design methods emerge. It is also shown which network class is best suited to support the reliability of the software and the whole system. After in introducing survey about the basics a realized circuit concept as an application is described together with program development aids for message-coupled dedicated microcomputer networks. With the help of this tool special applications can be implemented by means of "distributed programming" basing on "hardware capsulated software modules". The outlined kit system especially supports the reliability of software.

[Herlihy 80] Herlihy, Maurice and Barbara Liskov.
Communicating Abstract Values in Messages.
Technical Report 200, Massachusetts Institute of Technology, October, 1980.
Computation Structures Group Memo.

Abstract

Abstract data types have proved to be a useful technique for structuring systems. In large systems, however, it is sometimes useful to have different regions of the system use different representations for the abstract data values. This paper describes a technique for communicating abstract values between such regions. The method was developed for use in constructing distributed systems, where the regions exist at different computers, and the values are communicated over the network. As such, the method defines a call-by-value semantics. The method is also useful in non-distributed systems wherever call-by-value is the desired semantics. An important example of such a use is a repository, such as a file system, for storing long-lived data.

[Hertweck 78] Hertweck, F., E. Raubold and F. Vogt.
X.25 Based Process/Process Communication.
In *Proceedings, Computer Network Protocols*, pages C3-1 -- C3-22. Universite' De Liege, February, 1978.

Abstract

This paper describes an end-to-end protocol for interprocess communication based on the X.25 virtual channel protocol. The main (software) device to couple the operating system of a host computer to a communication network is the "Message Transmission Controller". Its structure and its principal functions are described. Special consideration is given to implementability on present day computer systems including pure host or simple terminal MTCs, but also host/front-end configurations. The problem process/process communication is mapped onto an interface process/MTC by defining a set of suitable interface commands to be executed by the process. The step to higher level (or application) protocols is done on the basis of the Communication Variable concept.

[Hewitt 77] Hewitt, Carl and Henry Baker.
Laws for Communicating Parallel Processes.
In *Proceedings, Information Processing 77*, pages 339-344. IFIP, 1977.

Abstract

This paper presents some laws that must be satisfied by computations involving communicating parallel

processes. The laws are stated in the context of the actor theory, a model for distributed parallel computation, and take the form of stating plausible restrictions on the histories of parallel computations to make them physically realizable. The laws are justified by appeal to physical intuition and are to be regarded as falsifiable assertions about the kinds of computations that occur in nature rather than as proven theorems in mathematics. The laws are used to analyze the mechanisms by which multiple processes can communicate to work effectively together to solve difficult problems.

Since the causal relations among the events in a parallel computation do not specify a total order on events, the actor model generalizes the notion of computation from a sequence of states to a partial order of events. The interpretation of unordered events in this partial order is that they proceed concurrently. The utility of partial orders is demonstrated by using them to express our laws for distributed computation.

[Huen 77] Huen, Wing, Peter Greene, Ronald Hochsprung, Ossama El-Dessouki.
A Network Computer for Distributed Processing.
In *Digest of Papers, COMPCON 77 Fall*, pages 326-330. IEEE, 1977.

Abstract

The TECHNEC is a network computer in the form of a ring of microcomputers (LSI-11s), designed for research in distributed processing. The design objectives, architecture and software support of the system are presented. Major user requirements such as pipelined compiling, automatic partitioning, and distributed control of machine intelligence applications are considered.

[Hunt 79] Hunt, J. G.
Messages in Typed Languages.
ACM-SIGPLAN Notices 14(1):27-45, 1979.

Abstract

Messages are increasingly being used for interprocess communication. The problem of introducing messages into typed languages is considered, and a solution in terms of typed message-channels is presented. Our particular treatment permits dynamic connections, including secure linking of separately-compiled programmes, and also features nondeterminacy, thereby enabling automatic resource-scheduling without monitors. Implementation considerations are discussed, and a comparison with the work of other authors is given.

[Hunt 80] Hunt, V. Bruce and Pier Carlo Ravasio.
Olivetti Local Network System Protocol Architecture.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 223-244. IBM, August, 1980.

Abstract

We describe the Olivetti Local Network System protocol architecture, which is a component of the Olivetti Local Network System architecture. The ISO open system interconnection reference architecture was used as a reference guide for the architecture. Our architecture provides the principal structures, attributes and component interfaces of the communication system to guide design and implementation of specific protocols. Fundamental mechanisms employed include a communication model, layering, and functional division. The communication model is based on an abstract communication primitive called a channel. The model is applicable at all levels in the hierarchy of layers. The architecture defines six layers including physical link, data link, transport, session, presentation, and application layers. Functional divisions specified are locator, transport, synchronization, error management , control and monitoring. Functional division is applied uniformly to each layer to achieve a coherent overall structure. Issues such as performance, name recognition, and flow control arising from the architecture's structure and associated implementation are discussed.

[Jacquemart 78] Jacquemart, Yves A.
Network Interprocess Communication in an X.25 Environment.
In *Proceedings, Computer Network Protocols*, pages C1-1 -- C1-6. Universite' De Liege, February, 1978.

Abstract

In this article we first define an interprocess communication facility in terms of service, access and function. After clarification of the X.25 principles we intend to use X.25 as the transmission service of the interprocess communication facility. We compare the X.25 service with other transmission services and we conclude by saying that a datagram transmission service beside X.25 is necessary.

[Jammel 80] Jammel, Alfons J., Pavel A. Vogel and Helmut G. Stiegler.
Impacts of Message Orientation.
In *Proceedings, IFIP Congress 80*, pages 281-286. IFIP, October, 1980.

Abstract

Two different basic operating system structures, procedure- and message-oriented, have been recognized. Strictly message-oriented systems are very rare, their direct distributability, however, strongly suggests that greater attention should be paid to them. Referring to the operating system BSM we illustrate and discuss the impact of a strictly message-oriented structure on parallelism, synchronization, protection, distributability, error recovery, and efficiency. No inherent handicaps due to message orientation have been encountered. Practical experience with BSM has led to the concept of manager processes. This concept seems to have contributed some new aspects to operating system design by making non-sequential processes manageable.

[Jazayeri 80] Jazayeri, Mehdi.
CSP/80: A Language for Communicating Sequential Processes.
In *Digest of Papers, COMPCON 80 Fall*, pages 736-740. IEEE, 1980.

Abstract

CSP/80 is a programming language intended for distributed applications. It is based on Hoare's communicating sequential processes (CSP). We discuss those aspects of CSP/80 that differ significantly from CSP and give the reasons why these departures from CSP were necessary.

[Jensen 78] Jensen, E. Douglas.
The Honeywell Experimental Distributed Processor-An Overview.
Computer 11(1):137-147, January, 1978.

Abstract

The Honeywell Experimental Distributed Processor (HXDP) is a vehicle for research in the science and engineering of processor interconnection, executive control, and user software for a certain class of multiple-processor computers which we call "distributed computer" systems. Such systems are very unconventional in that they accomplish total system-wide executive control in the absence of any centralized procedure, data, or hardware. The primary benefits sought by this research are improvements over more conventional architectures (such as multi-processors and computer networks) in extensibility, integrity, and performance. A fundamental thesis of the HXDP project is that the benefits and cost-effectiveness of distributed computer systems depend on the judicious use of hardware to control software costs. In this paper we describe the class of computer systems of interest to the HXDP project, the motivations for our interest, our research approach the initial application environment, the HXDP system philosophy, and the HXDP hardware facilities as seen by the executive programmer. The software portion of the executive will be described in a subsequent paper.,

[Johnson 75] Johnson, Paul R., Richard E. Schantz and Robert H. Thomas.
Interprocess Communication to Support Distributed Computing.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 199-203. ACM, March, 1975.

Abstract

A distributed computing system is, by definition, dependent upon communication between the distributed elements for its existence. It has become common to refer to each instance of parallel activity in a computer system as a process. Therefore, what is known as interprocess communication (IPC) is the lifeline or essential building block for any distributed computing facility. In the narrow sense, our concern with IPC is with the characteristics of a mechanism and interface which permit reliable communication of data between processes. In a much broader sense, IPC involves not only the facility for transmitting data, but also such questions as what gets transmitted and to whom, when it gets transmitted, what form it takes, and how it is used.

[Jones 79] Jones, Anita K., Robert J. Chansler Jr., Ivor Durham, Karsten Schwans and Steven R. Vegdahl.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 117-127. ACM, December, 1979.

Abstract

StarOS is a message-based, object-oriented, multiprocessor operating system, specifically designed to support *task forces*, large collections of concurrently executing processes that cooperate to accomplish a single purpose.

StarOS has been implemented at Carnegie-Mellon University for the 50 processor Cm* mulit-microprocessor computer. In this paper, we first discuss the attributes of task force software and of the Cm* architecture. We then discuss some of the facilities in StarOS that allow development and experimentation with task forces. StarOS itself is presented as an example task force.

[Joseph 81] Joseph, Mathai.
Schemes for Communication.
Technical Report 122, Department of Computer Science, Carnegie-Mellon University, June, 1981.

Abstract

This report describes features of a language for distributed and parallel programming which has been designed to provide flexibility in the transfer of information and control between the individual components of a program. The language allows synchronous and asynchronous message-passing, multiple-source input and broadcast output, and enables particular features of a distributed architecture to be efficiently accommodated without modification to the language. The module serves as the unit of encapsulation and a single communication takes place between an output *port* in one module and a set of input *ports* in other modules: each port has a *control rule* which specifies the protocol for sending or receiving messages, and is associated with a particular *communication scheme* which implements the communication operations. Modules are assumed to execute independently of each other except when they communicate by sending messages: the lifetime of a module is therefore limited only by its ability to send and receive messages. The use of the distinctive features of the language, such as broadcast mode output, is illustrated with several examples.

[Kain 76] Kain, Richard Y.
Seven Dimensions of Message Transmission Protocols.
[Unpublished Document].

Abstract

Message transmission protocols differ according to 1) whether or not the sender waits for an acknowledgement, 2) how the sender addresses the message, 3) how the receiver detects that a message exists, 4) whether the receiver selects the messages, 5) how the receiver identifies the sender, 6) how the receiver identifier the message, and 7) who determines the message lifetime. In each dimension the various options have different advantages. The choice of an option determines the kinds of errors that can cause non-functionality.

[Kain 80] Kain, Richard Y. and William R. Franta.
Interprocess Communication Schemes Supporting System Reconfiguration.
In *Proceedings, Computer Software and Application Conference*, pages 365-371. IEEE, October, 1980.

Abstract

Reliablitity in modular computer systems can be improved by redundancy. At the process level, this requires either the creation of standby processes and communications interconnections, or the provision of dynamic recovery mechanisms. This paper discusses the general reconfiguration problem, suggests four system designs for dealing with the problem, and presents evaluations of each in terms of modularity and reliability.

[Kieburtz 81] Kieburtz, Richard B.
A Distributing Operating System for the Stony Brook Multicomputer.
In *Proceedings, Second International Conference on Distributed Computing Systems*, pages 67-79. IEEE, 1981.

Abstract

The Stony Brook Multicomputer is hierarchially organized network of computer nodes that has been designed to support problem-solving by decomposition. High performance, relative to the speed of its individual processors, is one of its primary design goals. This paper describes the design of a message-based, distributed, operating system nucleus for the network. The nucleus of an operating system provides an interface between a physical machine and higher levels of software that implement abstract resources to be used by applications programs. Thus it is strongly influenced by the hardware architecture of a system. The design philosophy is to create levels of abstract machines, and to embed the necessary communication protocols into these abstract machines. The system supports a hierarchy of distributed file systems, with capability-based protection.

[Knight 81] Knight, Jeremy and Marty Itzkowitz.
THC -- A Simple High-Performance Local Network.
In *Proceedings, Second International Conference on Distributed Computing Systems*, pages 354-359. IEEE, April, 1981.

Abstract

We describe our need for a local network and the reasons we chose HYPERchannel as the hardware with which to implement it. We then present our reasons for choosing interprocess communication as the principle service of THC (The HYPERchannel Connection) and the design choices made in specifying the network. We then describe the structure and operation of the network. We then go on to describe the pseudocode technique used to complete the design and we briefly discuss the specific implementations for the various systems in our network. Finally we give performance measurements for the actual implemenation and present our conclusions.

[Knott 74] Knott, Gary D.
A Proposal for Certain Process Management and Intercommunication Primitives.
SIGOPS-Operating Systems Review 8(4), October, 1974.

Abstract

The notation of a process, and with it the possibilities for process intercommunication are fundamental in modern operating system design and, in disguised form, in porposals for languages which admit asynchrony. A straightforward repertoire of process controal and process intercommunication primitives are proposed and illustrated below. These primitives are interrupt-based. The general approach is founded upon the work of Brinch Hansen [B14], Walden [W1], and Bernstein et al [B7].

To begin, we shall elaborate on the notion of a process and sketch a process-processor relation to be used as a model. We note briefly that other operating system issues must be kept in mind. The various primitives are then described in detail, and following this, they are illustrated and compared with other proposals.

[Koch 82] Koch, A. and T. S. E. Maibaum.
A Message Oriented Language for System Applications.
In *Proceedings, Third International Conference on Distributed Computing Systems*, IEEE, ??, 1982.
[draft, maybe not accepted].

Abstract

The report outlines the design of an architecture independent programming language which takes advantage of the features of distributed computer architectures. To reflect the acceptance of the use of abstract data types in both the programming process and in language design, the language incorporates a mechanism for their implementation. This construct allows a programmer to write programs which use the operations of the type in parallel to any degree supported by the abstract properties of the type. The language also incorporates a mechanism for the "active" components of programs with the programmer being encouraged to regard this construct as a collection of functions (as opposed to the collection of operations for a data type). Powerful message passing mechanisms are incorporated into the language to provide a strictly typed, asynchronous mechanism for communication. Although we do not outline the ideas here, the language is supported by powerful design and analysis techniques.

[Kramer 81] Kramer, J., H. Magee and M. Sloman.
Intertask Communication Primitives for Distributed Computer Control Systems.
In *Proceedings, Second International Conference on Distributed Computer Systems*, pages 404-411. IEEE, April, 1981.

Abstract

This paper concentrates on the study of intertask communication primitives suitable for a distributed process control environment. The communication requirements are identified in terms of process control applications. The requirements for task behaviour, robustness and response time are described with respect to these transactions. Existing proposals for communication primitives are examined and found to be wanting. Finally, a set of primitives are proposed which match the requirements more satisfactorily than existing proposals.

[Lantz 80] Lantz, Keith A.
RIG, An Architecture for Distributed Systems.
In *Proceedings, Pacific '80*, ACM, November, 1980.

Abstract

At the University of Rochester we have had six years of experience in the design and implementation of a multiple-machine, multiple-network distributed system called RIG. RIG was built to serve as the sole intermediary between the human user (working through a display terminal or personal computer) and his available computer facilities. As far as possible, RIG attempts to present a coherent view of the distributed system similar to that provided by a traditional operating system for a single computer. The design of RIG is based on a model of distributed computation -- independent processes communicating only by messages -- which allows programmers to ignore the details of network and system configuration. The RIG Virtual Terminal Management System, togeather with a consistent command interaction discipline, allows the end-user to engage in multiple simultaneous activities and isolates him from the idiosyncrasies of each individual activity. This paper presents an overview of RIG, discusses some of its major successes, and suggests avenues for future research.

[Lauer 79] Lauer, Hugh C. and Roger M. Needham.
On the Duality of Operating System Structures.
SIGOPS-Operating Systems Review 13(2):3-19, April, 1979.

Abstract

Many operating system designs can be placed into one of two very rough categories, depending upon how they implement and use the notions of process and synchronization. One category, the "Message-Oriented System," is characterized by a relatively small, static number of processes with an explicit message system for communicating among them. The other category, the "Procedure-Oriented System," is characterized by a large, rapidly changing number of small processes and a process synchronization mechanism based on shared data.

In this paper, it is demonstrated that these two categories are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other. The principal conclusion is that neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built, not the application which the system will ultimately support.

[Le Lann 77] Le Lann, Gerard.
Distributed Systems--Towards a Formal Approach.
In *Information Processing 77*, IFIP, 1977.

Abstract

Packet-switching computer communication networks are examples of distributed systems. With the large scale emergence of mini and micro-computers, it is now possible to design special or general purpose distributed systems. However, as new problems have to be solved, new techniques and algorithms must be devised to operate such distributed systems in a satisfactory manner. In this paper, basic characteristics of distributed systems are analyzed and fundamental principles and definitions are given. It is shown that distributed systems are not just simple extensions of monolithic systems. Distributed control techniques used in some planned or existing systems are presented. Finally, a formal approach to these problems is illustrated by the study of a mutual exclusion scheme intended for a distributed environment.

[Liskov 79] Liskov, Barbara.
Primitives for Distributed Computing.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 33-42. ACM, December, 1979.

Abstract

Distributed programs that run on nodes of a network are now technologically feasible, and are well-suited to the needs of organizations. However, our knowledge about how to construct such programs is limited. This paper discusses primitives that support the construction of distributed programs. Attention is focused on primitives in two major areas: modularity and communication. The issues underlying the selection of the primitives are discussed, especially the issue of providing robust behavior, and various candidates are analyzed. The primitives will ultimately be provided as part of a programming language that will be used to experiment with construction of distributed programs.

[Liskov 81] Liskov, Barbara and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
Technical Report 210, Massachusetts Institute of Technology, November, 1981.
Computation Structures Group Memo.

Abstract

This paper presents an overview of an integrated programming language and system designed to support the construction and maintenance of distributed programs: programs in which modules reside and execute at

communicating, but geographically distinct, nodes. The language is intended to support a class of applications in which the manipulation and preservation of long-lived, on-line, distributed data is important. The language addresses the writing of robust programs that survive hardware failures without loss of distributed information and that provide highly concurrent access to that information while preserving its consistency. Several new linguistic constructs are provided; among them are atomic actions, and modules called guardians that survive node failures.

[Liu 77] Liu, Ming T. and Cecil C. Reames.
Message Communication Protocol and Operating System Design for the Distributed Loop Computer Network (DLCN).
In *Proceedings, Fourth Annual Symposium on Computer Architecture*, March, 1977.

Abstract
The Distributed Loop Computer Network (DLCN) is envisioned as a powerful, unified distributed computing system which interconnects midi/mini/micro- computers, terminals and other peripherals through careful integration of hardware, software and a loop communication network. Research concerning DLCN has concentrated on the loop communication network, message protocol and distributed network operating system. For the loop communication network, previous papers [2,3] reported a novel message transmission mechanism, its hardware implementation, and its superior performance verified by GPSS simulation. This paper presents an overview of the design requirements and implementation techniques for DLCN's message protocol and network operating system. Firstly, a bit-oriented distributed message communication protocol (DLMCP) which handles four message types under one common format is proposed. Besides user information transfer, this protocol supports automatic hardware-generated message acknowledgment, error detection and recover, and network control and distributed operating system functions. Secondly, the network operating system (DLOS) is described which provides facilities for interprocess communication by process name, global process control and calling of remote programs, generalized data transfer, alterable multi-linked process control structures, distributed resource management, and logical I/O transmission in a distributed file system.

[Liu 81] Liu, Ming T., Duen-Ping Tsay, Chuen-Pu Chou and Chun-Ming Li.
Design of the Distributed Double-Loop Computer Network (DDLCN).
Journal of Digital Systems 4(4), March, 1981.

Abstract
This paper presents the sytem design of the Distributed Double-Loop Computer Network (DDLCN), which is a fault-tolerant distributed processing system, that interconnects midi, mini, and micro computers using a double-loop structure. Several new features and innovative concepts have been integrated into the hardware, communications, software, and applications of DDCLN. The interface design is unique in that it employs tri-state control logic and bit-sliced processing, thereby enabling the network to become dynamically reconfigurable and fault-tolerant with respect to communication link failure as well as component failure in the interface. Three classes of N-process communication protocols, each providing a different degree of reliability, have been developed for exchanging multi-destination messages. Two synchronization mechanisms, eventcounts and sequencers (low-level) and control abstraction (high-level), are provided for use in distributed process synchronization. A new concurrency control mechanism, which uses distributed control without global locking and is deadlock-free, has been developed for use in distributed database systems. Finally, a distributed programming language called DISLANG has been proposed for use in implementing distributed systems software. The language uses a new concept, called Communicating Distributed Processes (CDP), to provide programmers with capabilities to handle specific problems in distributed computing environments, such as global operations, communication delay and failure, N-process communication, etc.

[Livesey 79] Livesey, Jon.
Inter-Process Communication and Naming in the Mininet System.
In *Digest of Papers, Eighteenth IEEE Computer Society International Conference*, pages 222-229. IEEE, February, 1979.

Abstract
We present a distributed message switched operating system, Mininet, in which inter-process communication is separated from object naming and protection.

All objects in the system are abstracted as executable objects, tasks, and inter-process communication is carried out between tasks without implying any particular method of object protection or naming. Naming and protection policies and mechanisms are implemented above the interprocess communication, and can be changed without changing it. In order to substantiate this, we present a particular model of resource naming and protection which

seems to fulfill the need to distribute resource access across the system, avoiding the need for centralized system control.

[Lorin 80] Lorin, Harold and Barry C. Goldstein.
Operating System Structures for Polymorphic Hardware.
Technical Report 35518, IBM, March, 1980.

Abstract

Given the technology that faces us, it is not unreasonable to project the existence of multiple processor configurations which have large numbers of processors with a variety of memory sharing and functional allocation possibilities.

This paper addresses some problems in the structure of operating systems that will manage such configurations so as to minimize the systems interference with application progress and provide for effective reaction to changing demands on the system. The structure of process creation and inter-process communication is explored in the context of two possible software/hardware structures.

[Manning 75] Manning, Eric and Richard W. Peebles.
Segment Transfer Protocols for a Homogeneous Computer Network.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 170-178. ACM, March, 1975.

Abstract

This research is focussed on solving certain problems of distributed processing on a distributed data base, with emphasis on transaction processing. Many data bases exhibit geographic locality of reference; most of the transaction homing on a given component of the data base originate from a particular geographic region. At the same time there is a need to operate the collection of components as a single data base, to provide for occasional transactions which cross regional boundaries, and for managerial queries and informantion retrieval applications which span the entire data base. There are many examples of this associated with business and industry; credit and inventory records for example. Finally, geographic locality of reference is only one of the reasons for creating logically unified but physically distributed data bases. If a data base contains information supplied by several agencies, each may insist as a matter of policy that "its" data be held in "its" hardware located on "its" premises, quite apart from the technical efficiencies which may accrue.

[Manning 77] Manning, Eric G. and Richard W. Peebles.
A Homogeneous Network for Data-Sharing Communications.
Computer Networks 1(2):211-224, 1977.

Abstract

The communications aspects of a distributed architecture for transaction processing are described. The architecture is aimed at transaction processing on physically distributed data bases, where most of the hits on a given component of the data base come from a single geographic region. The architecture is *physically* based on a homogeneous set of host minicomputers, a message-switched communications subnetwork (loop or packet- switched), and a set of network interface processors which connect the hosts to the communications subnetwork. It is *logically* based on two primitives; all data objects (including messages) are segments and all control objects (including messages) are *segments* and all control objects (including messages) are *tasks*. Each task runs in a private virtual space and all inter-task communication is done by passing message segments. Segment passing is done by a single message-switching task in each host, assisted by the interface processors and communications subnetwork where necessary. The message-switching task also enforces protection rules without the need for special hardware.

A two-host implementation of the logical architecture is operational. It is based on PDP-11 minicomputers and a non-switched wire pair subnetwork. The companion paper describes modelling studies of the architecture, using simulation and queueing-theoretic techniques.

[Manning 80] Manning, Eric, Jon Livesey and H. Tokuda.
Interprocess Communication in Distributed Systems: One View.
In *Proceedings, IFIP Congress 80*, pages 513-520. IFIP, October, 1980.

Abstract

This paper first describes the program of experimental research in distributed systems which has been carried out in the Computer Communications Networks Group of the University of Waterloo, over the past six years. The focus of the paper is on inter-process communication (IPC) techniques, and we therefore provide a comparison of

message-switched IPC facilities in several distributed systems developed both at Waterloo and elsewhere. The points of comparison include message management, synchronization modes, and performance. We have almost invariably chosen message-switched IPC for our distributed systems, and we examine the reasons for these decisions. Finally, we draw a few conclusions.

[Mao 80] Mao, T. William and Raymond T. Yeh.
Communication Port: A Language Concept for Concurrent Programming.
Transactions on Software Engineering SE-6(2):194-204, March, 1980.

Abstract

A new language concept-communication port (CP), is introduced for programming on distributed processor networks. Such a network can contain an arbitrary number of processors each with its own private storage but with no memory sharing. The procesors must communicate via explicit message passing. Communication port is an encapsulation of two language properties: "communication nondeterminism" and "communication disconnect time." It provides a tool for programmers to write well- structured, modular, and efficient concurrent programs. A number of examples are given in the paper to demonstrate the power of the new concepts.

[Metcalfe 72] Metcalfe, Robert M.
Strategies for Interprocess Communication in a Distributed Computing System.
In *Proceedings, Symposium on Computer-Communication Networks and Teletraffic*, Polytechnic Institute of Brooklyn, April, 1972.

Abstract

A recurring problem in the development of the ARPA Computer Network (ARPANET) is that of organizing the coordination of remote processes. ARPANET experience leads us to suggest that there are valuable distinctions to be made between: (1) *distributed* interprocess communication as required in computer network; and (2) *centralized* interprocess communication as often employed within computer operating systems. On the basis of a preliminary conceptualization, we propose that good strategies for *distributed* interprocess communication should be used *more* generally in computer operating systems because: (1) they have a clarifying effect on the management of multiprocess activity; and (2) they generalize well as operating systems themselves become more distributed.

[Miller 81] Miller, Barton and David Presotto.
XOS: An Operating System for the X-TREE Architecture.
SIGOPS-Operating Systems Review 15(2):21-32, April, 1981.

Abstract

This paper describes the fundamentals of the X-TREE Operating System (XOS), a system developed to investigate the effects of the X-TREE architecture on operating system design. It outlines the goals and constraints of the project and describes the major features and modules of XOS. Two concepts are of special interest: The first is demand paging across the network of nodes and the second is separation of the global object space and the directory structure used to reference it. Weaknesses in the model are discussed along with directions for future research.

[Mills 75] Mills, David L.
The Basic Operating System for the Distributed Computer Network.
Technical Report 416, University of Maryland, October, 1975.

Abstract

This report describes the Basic Operating System (BOS) for the Distributed Computer Network (DCN). The BOS is a multiprogramming executive providing process and storage management, interprocess communications, input/output device control and application-program support. It operates with any PDP11 model including at least 4K of storage, an operator's console and a communication device for connection to the DCN.

Included in this report is a description of the various components that make up the BOS and the manner in which they operate. Also described are the various primitive functions and command operations used to control the operation of the network and the various application programs. Other reports, listed in the references, describe the functioning of the DCN as a whole and also the upwardly-compatible Virtual Operating System (VOS) developed for PDP11 models with memory management features.

[Mills 76] Mills, David L.
An Overview of the Distributed Computer Network.
In *Proceedings, National Computer Conference*, pages 523-531. AFIPS, 1976.

Abstract

The Distributed Computer Network (DCN) is a resource-sharing computer network (DCN) which includes a number of DEC PDP11 computers. The DCN supports a number of processes in a multiprogrammed virtual environment. Processes can communicate with each other and interface with this environment in a manner which is independent of their residence within a particular computer. Resources such as processors, devices and storage media can be remotely accessed and shared so as to provide increased reliability, flexibility and system utilization.

The DCN now supports several programming languages and application packages. Programming languages such as SIMPL, LISP, BASIC and others, along with an extensive library of interactive graphics procedures, can be executed in processes which take full advantage of the distributed architecture of the network. Many of the components of the Disk Operating System (DOS) for the PDP11 can be executed in a special emulator-type virtual process now being constructed for this purpose. In this manner the PDP11 assembler, FORTRAN compiler and various system utilities can be supported in the network environment. In cases which exceed the processing power of the network, connections are available to two large Univac 1100-series machines.

[Morling 78] Morling, R. C. S., G. Neri, G. D. Cain, E. Faldella, T. Salmon, D. J. Stedham. The MININET Inter-Node Control Protocol. In *Proceedings, Computer Network Protocols* , pages B4-1 -- B4-6. Universite' De Liege, February, 1978.

Abstract

MININET is a packet-switching data transportation network being developed as a solution to the problem of local area data networking, with particular emphasis on low-cost interconnections for instrumentation environments. This paper describes a protocol for the interchange of messages that relate to the internal operation of the network, and which must be blended unobtrusively into the packet streams being transported between the users of the network. Details of the differences between user and network packet structures and handling are presented and it is shown that adoption of a half duplex version of a protocol developed earlier, the MININET Link Protocol, preserves the essential simplicity of that protocol and satisfies the requirements for internal network conversations.

[Morris 72] Morris, D., G. R. Frank and T. J. Sweeney. Communications in a Multi-Computer System. In *Proceedings, Conference on Computers-Systems and Technology*, pages 405-414. The Institution of Electronic and Radio Engineers, October, 1972.

Abstract

The MU5 system being constructed at the University of Manchester consists of several computers connected together so that they may access each other's stores. The operating system for the complex is sub-divided into about 16 separate programs which run independently except for communicating with each other via a formalised message-switching system. These programs are distributed across the machines of the complex hence the message-switching systems of the separate machines are linked. Within one machine messages are transferred by passing pointers to page tables rather than by copying the information. Transfers between machines and copying pages as necessary.

[Nelson 80] Nelson, Bruce Jay. Remote Procedure Call. PhD Thesis Proposal, Department of Computer Science, Carnegie-Mellon University.

Abstract

Remote procedure call is the transfer of control between programs in disjoint address spaces which share no resources except a narrow communication medium.

This proposal first establishes a perspective for the work and goals of the thesis. We then define remote procedures precisely, outline the important issures with an example, and characterize the benefits. We survey past work on remote procedures-- briefly commenting on its relationship to message-passing systems-- and examine some existing implementations. These efforts are shown to be weak when measured against a spectrum of important remote procedure issues: strong typechecking, parameter functionality, binding and configuration, exactly-once semantics, and error handling and crash recovery. We discuss these issues in detail and propose those which the thesis will investigate in depth. Some preliminary results on configuration are given.

[Nelson 81] Nelson, Bruce Jay.
Remote Procedure Call.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, May, 1981.

Abstract

Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel. The thesis of this dissertation is that remote procedure call (RPC) is a satisfactory and efficient programming language primitive for constructing distributed systems.

A survey of existing remote procedure mechanisms shows that past RPC efforts are weak in addressing the five crucial issues: uniform call semantics, binding and configuration, strong typechecking, parameter functionality, and concurrency and exception control. The body of the dissertation elaborates these issues and defines a set of corresponding *essential properties* for RPC mechanisms. These properties must be satisfied by any RPC mechanism that if fully and uniformly integrated into a programming language for a homogeneous distributed system. Uniform integration is necessary to meet the dissertation's fundamental goal of syntactic and semantic *transparency* for local and remote procedure. Transparency is important so that programmers need not concern themselves with the physical distribution of their programs.

In addition to these essential language properties, a number of *pleasant properties* are introduced that ease the work of distributed programming. These pleasant properties are good performance, sound remote interface design, atomic transactions, respect for autonomy, type translation, and remote debugging.

With the essential and pleasant properties broadly explored, the detailed design of an RPC mechanism that satifies all of the essential properties and the performance property is presented. Two design approaches are used: The first assumes full programming language support and involves changes to the language's compiler and binder. The second involves no language changes, but uses a separate translator- a source-to-source RPC compiler- to implement the same functionality.

[Ousterhout 79] Ousterhout, John K., Donald A. Scelza and Pradeep S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure (Summary).
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 115-116. ACM, December, 1979.

Abstract

The paper is a discussion of the issues that arose in the design of an operating system for a distributed multiprocesor, Cm*. Medusa is an attempt to understand the effect on operating system structure of distributed hardware, and to produce a system that capitalizes on and reflects the underlying architecture. The resulting system combines several structural features that make it unique among existing operating systems.

[Ousterhout 80a] Ousterhout, John K.
Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, April, 1980.

Abstract

This dissertation is an analysis of the design of Medusa, an operating system with a highly distributed control structure that runs on the Cm* multimicroprocessor. In order to gain an understanding of how to exploit distributed hardware, the system's structure was allowed to derive directly from the constraints of the underlying machine. The Cm* hardware is distributed, yet extremely flexible in the kinds of interprocessor communication it permits. Thus Medusa's structure arose from a consideration of two issues: partitioning and cooperation. How should the system be partitioned in order to enhance its modularity and make use of the distributed hardware How should the separate subunits communicate so as to function together in a robust way as a single logical entity The resulting system combines several structural features that make it unique among existing operating systems.

In order to provide modularity and to capitalize on the distributed hardware, Medusa consists of five relatively independent utilities that execute on different processors. Each utility provides one abstraction for the rest of the system and communicates with user programs and other utilities via messages. Functions are distributed between utilties at a very low level (for example, no one utility contains enough functionality to create and execute a new program without assistance from other utilities). The message communication mechanism plays a central role in the system; it is discussed in detail and compared to other existing or proposed mechanisms. The

distribution of the utilities presents a deadlock danger. It is shown how a coroutine-based utility structure avoids deadlock.

[Ousterhout 80b] Ousterhout, John K., Donald A. Scelza and Pradeep S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.

Abstract

The design of Medusa, a distributed operating system for the Cm* multimicroprocessor, is discussed. The Cm* architecture combines distribution and sharing in a way that strongly impacts the organization of operating systems. Medusa is an attempt to capitalize on the architectural features to produce a system that is modular, robust, and efficient. To provide modularity and to make effective use of the distributed hardware, the operating system is partitioned into several disjoint utilities that communicate with each other via messages. To take advantage of the parallelism present in Cm* and to provide robustness, all programs, including the utilities, are task forces containing many concurrent, cooperating activities.

[Panzieri 82] Panzieri, F. and S. K. Shrivastave.
Reliable Remote Calls for Distributed UNIX: An Implementation Study.
In *Proceedings, Second Symposium on Reliability in Distributed Software and Database Systems*, pages 127-133. July, 1982.

Abstract

An implementation of a reliable remote procedure call mechanism for obtaining remote services is described. The reliablity issues are discussed together with how they have been dealt with. The performance of the remote call mechanism is compared with that of local calls. The remote call mechanism is shown to be an efficient tool for distributed programming.

[Panzieri ??] Panzieri, F. and S. K. Shrivastava.
The Design of a Reliable Remote Procedure Call Mechanism.
[The University of Newcastle upon Tyne - Computing Laboratory].

Abstract

Starting from the hardware level that provides primative facilities for data transmission, we describe how a reliable Remote Procedure Call mechanism can be constructed. We discuss various design issues involved, these include the choice of a message passing system over which the remote procedure call mechanism is to be constructed and the treatment of various abnormal situations such as lost messages and node crashes.

[Pardo 78] Pardo, Roberto, Ming T. Liu and Gojko A. Babic.
An N-Process Communication Protocol for Distributed Processing.
In *Proceedings, Symposium on Computer Network Protocols*, pages 13-15. IEEE, February, 1978.

Abstract

A Distributed Processing Algorithm (DPA) is an algorithm whose execution involves interaction between two or more remote processes in a distributed processing system. Most of the software issues in distributed processing systems are related to the concept of DPA's. One important aspect is the message exchange (protocol) requirements induced by the DPA's. Current high-level communication protocols efficiently support the establishment, maintenance, and termination of connections between two processes, and thus can be called 2-process communication protocols. However, this class of protocols limits the type of DPA's that can be efficiently supported by a distributed processing system. In this paper we propose a class of protocols that are not constrained to handle only 2-process communication but rather any "network of connections," and we refer to a protocol in this class as an n-processs communication protocol. The purpose of this paper is to motivate the need for such protocols, to show their relationship with distributed processing systems, and to establish their features.

[Peberdy ??] Peberdy, N. J.
Distributed Computer Systems - A Model.
[unknown, pages 17-25].

Abstract

The past five years have seen a dramatic changeabout in traditional hardware/software relationships: hardware costs have plummeted, and the size, environmental requirements and reliability of computing elements have altered drastically. It now becomes feasible to distribute a computing system, such that processors may be placed adjacent to the processes they control. These distributed computing modules operate in an essentially parallel

mode, but are required to communicate in order to co-ordinate their activities. Reliable, secure communication systems must be established to ensure correct operation. Such systems are not only functions of the electrical hardware employed, but also of the software support provided. Of vital importance are the protocols selected, which define and detail an agreed procedure for the exchange of information

This paper reviews the fundamental software considerations in the design of computer networks, with specific relevance for process-control applications. It discusses in detail, inter-connection strategies and protocols and briefly examines currently adopted schemes. The implications of fully decentralized system control are considered. Of particular concern is the question of the production of reliable, fault-tolerant, secure systems.

[Peebles 78] Peebles, Richard and Eric Manning.
System Architecture for Distributed Data Management.
Computer 11(1):40-47, January, 1978.

Abstract

Successful implementation of most distributed processing systems hinges on solutions to the problems of data management, some of which arise directly from the nature of distributed architecture, while others carry over from centralized systems, acquiring new importance in their broadened environment. Numerous solutions have been proposed for the most important of these problems.

In a distributed computer system, multiple computers are logically and physically interconnected over "thin-wire" (low bandwidth) channels and cooperate under decentralized system-wide control to execute application programs. Examples of thin-wire systems are Arpanet, the packet-switched network of the U.S. Defense Communications Agency, and Mininet, a transaction-oriented research network being developed at the University of Waterloo. These may be contrasted with high-bandwidth or "thick-wire" multiprocessor architectures, such as the Honeywell 6080 or the Pluribus IMP. A practical consequence of thin-wire design is that processing control is in multiple centers. No one processor can coordinate the others; all must cooperate in harmony as a community of equals.

The key issue is that interprocess communication is at least an order of magnitude slower when the communicating tasks are in separate computers than it is when they are executing in the same machine. Therefore, no single process can learn the global state of the entire system nor issue control commands quickly enough for efficient operation, so that multiple centers of control are implied.

[Pouzin 73] Pouzin, Louis.
Presentation and Major Design Aspects of the Cyclades Computer Network.
In *Proceedings, Third Data Communications Symposium*, pages 80-87. IEEE and ACM, November, 1973.

Abstract

A computer network is being developed in France, under government soonsorship, to link about twenty heterogeneous computers located in universities, research and D.P. Centers. Goals are to set up a prototype network in order to foster experiment in various areas, such as: data communications, computer interaction, cooperative research, distributed data bases. The network is intended to be both, an object for research, and an operational tool.

In order to speed up the implementation, standard equipment is used, and modifications to operating systems are minimized. Rather, the design effort bears on a carefully layered architecture, allowing for a gradual insertion of specialized protocols and services tailored to specific application and user classes.

[Pouzin 75a] Pouzin, Louis.
Virtual Call Issues in Network Architectures.
Technical Report SCH 559.1, Institut de Recherche d'Informatique et d'Automatique (IRIA), September, 1975.

Abstract

The concept of virtual circuit is mainly used to designate a set of end-to-end control mechanisms in packet switching networks. Similar mechanisms called liaisons may be found at higher levels in a computer network. Their properties are reviewed, specifically with regard to port access, error and flow control. Various forms of virtual circuits are included in existing or planned packet networks. But some networks have none. Since end-to-end control mechanisms always exist at higher levels, it is not clear that virtual circuits in packet networks are worth their cost.

Interfacing computer systems with virtual circuits raises a number of problems specifically in splicing with liaisons

at a higher level. Another approach is a gateway mimicking terminals. Finally, the least interfering approach is to consider virtual circuits as a substitute to real ones.

[Pouzin 75b] Pouzin, Louis.
An Integrated Approach to Network Protocols.
Technical Report NCP 500.1, Institut de Recherche d'Informatique et d'Automatique (IRIA), May, 1975.

Abstract
Host-to-host protocols (H-H) for heterogeneous computer networks are still in infancy. So far very few implementations are in existence. Among those on which documentation is available are Arpanet and Cyclades. The former provides only for basic services allowing the transfer of up to 1000 octet messages, with flow control but not error control. The latter allows up to 32000 octet messages, with error and flow control. Both are similar in the sense that they offer only a message transfer service, which is intended for building higher level protocols more appropriate for specific uses. Since data to be transfered are usually structured in various ways, a traditional approach is to superimpose additional layers of specific protocols, each one dealing with a particular level of structure. While being functionally correct this approach leads to heterogeneity, redundancy and overhead among the various layers.

[Pouzin 76] Pouzin, Louis.
Virtual circuits vs. datagrams -- Technical and political problems.
Technical Report SCH 576.1, Institut de Recherche d'Informatique et d'Automatique (IRIA), June, 1976.

Abstract
Public packet networks are becoming a reality, and call for interface standards. Two levels of facilitcs have been proposed, virtual circuit (VC), and datagram (DG). The concepts of VC and DG are already well developed within computer networks. Their properties are reviewed, along with typical issues such as out-of-sequence and congestion problems.

Usually DG's are a sub-layer used as a transport facility by a VC protocol. They also provide the ability to extend switching functions within user systems. The characteristics of VC's considered by CCITT are examined critically, and related to experimental networks and manufacturer softwares.

VC's and DG's are compared from the viewpoint of adapting customer systems to public networks. When the customer is interested in a transport facility, DG's appear to have an edge. When a network becomes a terminal handler, adaptations are more complex and require character stream interfaces. Intelligent teminals would make this problem disappear, as they can use a DG interface.

Although various groups call for a DG interface, the carriers are opposed to it. Four carriers are rushing a VC protocol through CCITT. The carrier's goal is to take over termnial handling, and gradually other processing functions. DG's would leave too much freedom to the customer. The political implications of the carrier policy suggest that better boundaries be drawn up between carriers and data processing.

[Proebster 78] Proebster, W. E. and V. Sadogopan.
Communication Technology and Concepts: Technical Status and Outlook.
Communication Technology 31966, IBM, December, 1978.

Abstract
The most important communication concepts are described in a systematic way, covering the hierarchy of communication elements, systems and services.

Current technical implementations of these concepts are discussed and the major future trends are highlighted. Special emphasis is given to fiber optics, communication satellites, large scale integration and microprocessors.

[Quatse 72] Quatse, Jesse T., Pierre Gaulene and Donald Dodge.
The External Access Network of a Modular Computer System.
In *Proceedings, Spring Joint Computer Conference*, pages 118-125. IEEE, 1972.

Abstract
A modular time-sharing computer system, called PRIME, is currently under development at the University of California, Berkeley. Basically, PRIME consists of sets of modules such as processors, primary memory modules, and disk drives, which are dynamically reconfigured into separate subsystems. One ramification of the architectural approach is the need for a medium to accommodate three classes of communications: (1) those

between any processor and any other processor, (2) those between any processor and any disk drive, external computer system, or other device in the facility pool, and (3) those between primary memory and any device in the facility pool. This paper describes the External Access Network (EAN) which was developed for this purpose. The EAN is specialized by certain PRIME implementation constraints. Otherwise, it is adaptable to any system having similar design objectives, or to aggregates of independent computer systems at the same site, which share a similar facility pool and which require system to system communications.

[Rao 80] Rao, Ram.
Design and Evaluation of Distributed Communication Primitives.
In *Proceedings, ACM Pacific '80*, pages 14-23. ACM, November, 1980.

Abstract

Communication primitives suitable for use in a distributed programming environment are the focus of this paper. The design of such primitives involves issues such as communication model, synchronization, selective receiving, message length, naming and buffering. Alternatives for handling these design issues are presented and their mutual dependencies are defined. A number of existing primitives are examined in the light of these design decisions.

Two benchmark problems are defined and programs for them have been written using each set of primitives. The programming experience is used to evaluate the privitives and draw conclusions about the design of communication primitives for various distributed programming environments.

[Rao 82] Rao, Ram.
A Kernel for Distributed and Shared Memory Communication.
Technical Report 82-06-01, Department of Computer Science, University of Washington, June, 1982.

Abstract

Interprocess communication via shared memory has received considerable attention in the past. More recently, there has been a growing interest in communication in distributed environments. This dissertation examines distributed communication and attempts to intergrate it with shared memory communication. ' A kernel is presented which provides simple tools to facilitate communication in these environments, and allows definition of new communication mechanisms. The kernel consists of features for synchronization and data transfer and locking. By combining the synchronization and data transfer facilities, distributed communication may be modelled. Shared memory communication normally requires synchronization and locking. Some applications require both shared memory and distributed communication. Such "hybrid" applications typically use the kernel features for synchronization, data transfer and locking. The kernel operations, though somewhat low level, provide flexibility in designing efficient mechanism well suited for specific applications. Several examples illustrate the use of the kernel in programming solutions to a variety of communication problems, as well as in modelling some programming language mechanisms (including Ada and CSP).

[Rashid 80] Rashid, Richard F.
An Inter-Process Communication Facility for UNIX.
Technical Report 124, Department of Computer Science, Carnegie-Mellon University, February, 1980.

Abstract

An inter-process communication facility implemented at Carnegie-Mellon University for VAX/UNIX version seven is described. This facility was designed to provide language, operating system and machine independent communication between processes performing distributed computations. Its relationships to previously existing UNIX facilities and other systems for distributed computing are discussed.

[Rashid 81] Rashid, Richard F. and George G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
Technical Report 123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

Abstract

Accent is a communication oriented operating system kernel being built at Carnegie-Mellon University to support the distributed personal computing project, Spice, and the development of a fault-tolerant distributed sensor network (DSN). Accent is built around a single, powerful abstraction of communication between processes, with all kernel functions, such as device access and virtual memory management accessible through messages and distributable throughout a network. In this paper, specific attention is given to system supplied facilities which

support transparent network access and fault-tolerant behavior. Many of these facilities are already being provided under a modified version of VAX/UNIX. The Accent system itself is currently being implemented on the Three Rivers Corp. PERQ.

[Rashid 82] Rashid, Richard F.
Accent Kernel Interface Manual.
Technical Report ??, Department of Computer Science, Carnegie-Mellon University, 1982.

Abstract

Accent is a communication oriented operating system kernel being built at Carnegie-Mellon University to support the distributed personal computing project, Spice, and the development of a fault-tolerant distributed sensor network (DSN). Accent is built around a single, powerful abstraction of communication between processes, with all kernel funcations, such as device access and virtual memory management accessible through messages and distributable throughout a network. In this manual, specific attention is given to the program interface to the Accent kernel. Many of the facilities described (in particular IPC related facilities) are already being provided under a modified version of VAX/UNIX. The Accent system itself is currently being implemented on the Three Rivers Corporation PERQ.

[Redell 80] Redell, David D., Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray and Stephen C. Purcell.
Pilot: An Operating System for a Personal Computer.
Communications of the ACM 23(2):81-92, February, 1980.

Abstract

The Pilot operating system provides a single-user, single-language environment for higher level software on a powerful personal computer. Its features include virtual memory, a large "flat" file system, streams, network communication facilities, and concurrent programming support. Pilot thus provides rather more powerful facilities than are normally associated with personal computers. The exact facilities provided display interesting similarities to and differences from corresponding facilities provided in large multi-user systems. Pilot is implemented entirely in Mesa, a high level system programming language. The modularization of the implementation displays some interesting aspects in terms of both the static structure and dynamic interactions of the various components.

[Reed 80] Reed, David and Liba Svobodova.
SWALLOW: A Distributed Data Storage System for a Local Network.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 355-373. IBM, August, 1980.

Abstract

SWALLOW is an experimental project that will test feasability of several advanced ideas on the design of object-oriented distributed systems. Its purpose is to provide a reliable, secure and efficient storage in a distributed environment consisting of many personal machines and one or more shared data storage servers. SWALLOW implements a uniform interface to all objects accessable from a personal computer: these objects can be stored either on the local storage device or in one of the data storage servers. The data storage servers provide stable, reliable, and long-term storage. The access control to objects in the data storage servers is based on encrypting the data; encryption is used to prevent both unauthorized release of information and unauthorized modification. SWALLOW can handle efficiently both very small and very large objects and it provides mechanisms for updating of a group of objects at one or more physical nodes in a single atomic action.

[Reid 80] Reid, Lorretta Guarino.
Control and Communication in Programmed Systems.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, September, 1980.

Abstract

The paper "On the Duality of Operating Systems Structures" by Lauer and Needham (1978) was an extremely controversial paper. It claimed to have demonstrated an important result about communication in operating systems, but it left many people uneasy and unsure of exactly what had been demonstrated. Attempts to formalize the results of the paper by casting it in terms of known models of systems failed, primarily because the models lacked the ability to represent the dynamic nature of systems.

A model of communication, consisting of primitive objects and communication operations, is developed in this

thesis in order to study communication properties of systems. Some of the goals of the design of the model were to permit us to deal with the dynamic recreation and destruction of pieces of systems, to permit the description of systems programmed in a wide variety of languages and implemented on a wide range of architectures, and to provide some support for flexibility and the localization of communication knowledge in systems.

Although the primitives of the model are sufficient to describe communication in systems, working with them directly is much like programming only with GO TO's. There is a lot of structure to the way that communications takes place, and this structure is not explicitly visible in the use of primitives. The thesis introduces a notation for describing *abstract communication constructs* in a way that permits the structure to be expressed precisely and that allows constructs to be compared for their similarities and differences.

The thesis uses the model and the notion of abstract communication constructs to explore several issues in communication. In an effort to characterize the properties of communication that make it easy to use programs in many different systems, the criterion of the flexibility of an implementation is developed. The Lauer-Needham paper is discovered not to demonstrate the duality of the two types of operating systems but to introduce an abstract communication construct that is flexible enough to be implemented directly in both systems. Finally, a proof is given of the necessary and sufficient conditions on communication for a system to be completely sequential.

[Retz 75] Retz, David L.
Operating System Design Considerations for the Packet-Switching Environment.
In *Proceedings, National Computer Conference*, pages 155-160. AFIPS, 1975.
Volume 44.

Abstract

One of the striking developments in computing and communication technology during the past decade is reflected in the evolution of packet-switching computer networks. Packet-switching communication techniques allow dynamic allocation of a set of communication resources (circuits) so that they may be flexibly shared among a number of autonomous processors. Implementation of such packet-switching networks has required many design decisions, such as the choice of network topology, routing strategies, and the establishment of conventions, or protocols, for information interchange between network resources.

This paper is concerned with the design requirements of Host operating systems: those systems whose primary business is the management of computing resources rather than communication resources. Low-level communication tasks such as routing fall outside the realm of the Host responsibilities discussed here and are performed by means of a sub-network of small computers dedicated of the task of packet-switching. In the ARPANET these computers are called Interface Message Processors, or IMPs, and use packet-switching techniques to communicate via 50-kilobit common carrier circuits. Each IMP provides up to four high-speed synchronous serial ports to which Hosts connect using special-purpose Host-IMP interfaces. Packet-switching network environments place special requirements on the design of the connected Host operating systems. Attachment to the ARPANET, for example, has required a number of additions or modifications to existing operating systems. There are certain structural features which must be incorporated in system design in order to facilitate effective use of distributed computing resources. We begin by examining a few of these features.

[Rowe 75] Rowe, Lawrence A.
The Distributed Computing Operating System.
Technical Report 66, Department of Information and Computer Science, University of California, Irvine, June, 1975.

Abstract

The Distributed Computing System (DCS) is a computer network architecture emphasizing reliable, fail-soft service of an operating system for a DCS. Issues discussed include interprocess communication, system initiation, and failure detection and recovery. Features of the implementation of a prototype system and some experiences gained from building and using the prototype are also described.

Conclusions made from this work are that problems and solutions discovered while developing minicomputer networks are the same as those encountered in developing networks of larger machines. Specifically, DCS and its operating system demonstrate that systems without centralized control can be constructed, that broadcast messages are useful, and that messages which are sent to a process but are intercepted and acted upon by the environment of the receiving process are necessary to achieve location independence.

[Ruschitzka 73] Ruschitzka, M. G. and R. S. Fabry.
The Prime Message System.
In *Digest of Papers, COMPCON 73*, pages 125-128. IEEE, February, 1973.

Abstract

The message system of the PRIME system which is currently being constructed at the University of California at Berkeley combines the addressing generality of a network message system with a cost conscious implementation typical of single processor systems. This paper deals with its design and implementation details.

[Ryan 79a] Ryan, M. D.
Design of a Distributed System: Overview of System.
The Australian Computer Journal 11(3):98-102, August, 1979.

Abstract

When consideration is given to distributed processing many different ideas are encountered. This is due to the fact that there are no clear concepts involved and much of the effort has been hardware rather than software driven. However, there is one clear thing about distributed processing and that is it is about communication, and for any application to be implemented there must be a solid basis of communication on which to build. This project is to do just that.

The basic design was started after a reasonable survey of the literature was carried out, however this led to the biggest problem of all, which was the separation of the concepts involved into the relevant areas. This led to a great deal of wasted effort. Despite this approach certain biases have had a great influence, hopefully for the better, on the final design.

[Ryan 79b] Ryan, M. D.
Design of a Distributed System: Interprocess Communication.
The Australian Computer Journal 11(3):103-107, August, 1979.

Abstract

In this paper interprocess communication is discussed within the constraints outlined in the companion paper (Ryan, 1979). The companion paper discusses the overall concepts of a distributed system and emphasizes the role played by interprocess communication in such systems. However, it asserts that interprocess communication should be designed within the host operating system and then extended to a network environment.

[Saettone 78] Saettone, R.
M I T S : Microprocessor Implementation of a Transport Station.
In *Proceedings, Computer Network Protocols Symposium*, pages E3-1 -- E3-5.
Universite' De Liege, February, 1978.

Abstract

This paper describes the implementation of the communication interface and transport protocol in the link between the CYCLADES network and a host computer, such as the IBM 360/67 at the University of Grenoble.

System throughout is increased by a multi-microprocessor architecture that executes not only the communication functions associated to a serial data link, but also most of the functions of the transport protocol used at the front of the network. It interfaces on one side with a serial, synchronous full duplex line via a modem. On the other side, it is attached to the channel of the host's I/O processor, as a peripheral device.

The main goal of this approach is to relieve the host from all the communication functions and to execute them in a functionally equivalent peripheral device. A considerable reduction in the cost/performance ratio is obtained by the use of general purpose microprocessors instead of a front-end mini-computer or a special-purpose processor.

[Sakai 77] Sakai, Toshiyuki, Tsunetoshi Hayashi, Shigeuoshi Kitazawa, Koichi Tabata and Takeo Kanade.
Inhouse Computer Network Kuipnet.
In *Proceedings, Information Processing 77*, pages 161-166. IFIP, 1977.

Abstract

The inhouse resource sharing computer network KUIPNET (Kyoto University Information Processing NETwork) is described. It is intended to support advanced researches in information processing by sharing resources such as files and devices among participating host computers in one building. The network can handle raw data such as digitized image and speech signals as well as character-oriented message data. Design consideration of the network and the operating systems of host computers are described. Some examples of applications using the

network are presented as well as results of traffic measurement.

[Schantz 75] Schantz, Richard E.
A Commentary on Procedure Calling as a Network Protocol.
Technical Report RFC # 684, ARPA Network Working Group, April, 1975.

Abstract

While the Procedure Call Protocol (PCP) and its use within the National Software Works (NSW) context attacks many of the problems associated with integrating indepenent computing systems to handle a distributed computation, it is our feeling that its design contains flaws which should prevent its widespread use, and in our view, limit its overall utility. We are not voicing our objection to the use of PCP, in its current definition, as the base level implementation vehicle for the NSW project. It is already too late for any such objection, and PCP may, in fact, be very effective for the NSW implementation, since they are proceeding in parallel and probably influenced each other. Rather, we are voicing an objection to the "PCP philosophy", in the hope of preventing this type of protocol from becoming the de-facto network standard for distributed computation, and in the hope of influencing the future direction of this and similar efforts.

[Schlichting 82] Schlichting, Richard D. and Fred B. Schneider.
Using Message Passing for Distributed Programming Proof Rules and Disciplines.
Technical Report TR 82-491, Department of Computer Science, Cornell University, May, 1982.

Abstract

Inference rules for proving the partial correctness of concurrent programs that use message-passing for synchronization and communication are derived. Three types of message-passing primitives are considered: synchronous, asynchronous and remote procedure call (rendezvous). The proof rules show how interference can arise and be controlled. They also provide insight into why distributed programs are hard to design and understand.

[Schmid 74] Schmid, Hans Albrecht.
An Approach to the Communication and Synchronization of Processes.
In *Proceedings, 1973 International Computing Symposium*, pages 165-171. IFIP, April, 1974.

Abstract

For the communication of concurrent processes we introduce primitives which allow uniform modelling of competition for devices, as well as of cooperation which taken place by exchange of synchronization signals. Using these primitives, process systems are split up into processes independent of, and processes communicating with the environment. This allows easy transformation of process systems into petri nets. Petri nets, as an abstract mathematical tool, seem to be appropriate to the treatment of all problems caused by interaction of concurrent processes, as for example deadlocks and their prevention.

[Sherman 82] Sherman, Richard H., Melvin G. Gable and Anthony Chung.
Overcoming Local and Long-Haul Incompatibility.
In *DATA COMMUNICATIONS*, pages 195-206. March, 1982.

Abstract

There are long-distance data networks-composed of switched or leased facilities,point-to-point, or multipoint connections--and now there are local networks. Most agree that the two will have to interconnect, and some progress is being made in this area. But it remains unclear how this can be done while retaining end-to-end network efficiency, reliability, connectivity, and cost-effectiveness.

A network protocol layer is needed that can adapt to the evolution, operation, and interconnection of such diverse networks. The network should accommodate computers that implement different network protocols, and the network components, such as interfaces and computers, should be as easy to install as modems--without requiring communications or computer specialists. Some modems, for example, can now sense the data rate and modulation scheme and automatically adapt. Whole networks should be able to merge with or separate from other networks as easily as individual network components are added and removed.

In view of this, an experimental network has been developed at Ford in an attempt to implement these evolutionary and operational objectives. Different types of networks were interconnected using a uniform network protocol layer developed to perform measurement and control functions.

[Shoch 79] Shoch, John F.
An Overview of the Programming Language Smalltalk-72.
ACM-SIGPLAN Notices 14(9):64-73, September, 1979.

Abstract

Smalltalk is a programming language designed around a single metaphor-- that similar objects can be grouped into more general classes. Starting with a conceptually elegant and consistent epistemology, it has been possible to construct a language with powerful semantic capabilities, while retaining a simple syntactic representation.

The language development itself is but one part of a broader effort to explore the ways in which people can manipulate information and communicate with machines. It is one tool utilized in the construction of an interactive computer system, used by both children and adults for problem solving, simulation, drawing and painting, real time generation of music, information retrieval, and other tasks.

[Silberschatz 81] Silberschatz, Abraham.
A Note on the Distributed Program Component Cell.
ACM-SIGPLAN Notices 16(7), July, 1981.

Abstract

This paper presents a new language construct for distributed computing. This construct, called a cell, allows one to simulate a variety of language constructs. Its salient features provide the programmer with an effective synchronization scheme, and a mechanism to control the order in which various activities within a cell should be executed.

[Sloman 80] Sloman, M. S. and S. Prince.
Local Network Architecture for Process Control.
In *Proceedings, IFIP Working Group 6.4 International Workshop on Local Networks for Computer Communications*, pages 407-427. IBM, August, 1980.

Abstract

The physical distribution of equipment and machinery on an industrial site makes it particularly suitable for implementing distributed computer control systems. There is also a need for a serial communication system even in a centralised control so as to save on wiring costs, which can be substantial.

This paper identifies the communication requirements for Distributed Process Control Systems and indicates the main differences between Process Control and other application areas.

A network architechture for Process Control which caters for arbitrary point-to-point or broadcast data links is presented. The architecture is based on the lower 4 layers of the ISO Open Systems Model. The services provided and functions performed by each layer is described. Network management is also briefly discussed.

[Solomon 79] Solomon, Marvin H. and Raphael A. Finkel.
The Roscoe Distributed Operating System.
In *Proceedings, Seventh Symposium on Operating Systems Principles*, pages 108-114. ACM, December, 1979.

Abstract

Roscoe is an operating system implemented at the University of Wisconsin that allows a network of microcomputers to cooperate to provide a general-purpose computing facility. After presenting an overview of the structure of Roscoe, this paper reports on experience with Roscoe and presents several problems currently being investigated by the Roscoe project.

[Spector 81a] Spector, Alfred Z.
Multiprocessing Architectures for Local Computer Networks.
Technical Report STAN-CS-81-874, Department of Computer Science, Stanford University, August, 1981.
[cute title].

Abstract

This dissertation discusses the interconnection of computers with very high speed local networks in a manner that can support a large class of distributed programs -- a class that includes programs requiring highly efficient interprocessor communication. This research is motivated by {1} prospects for local networks having a capacity of 100 megabits/second or higher; {2} continuing advances in semiconductor technology; {3} the increasing availability of inexpensive, low-latency, non-volatile storage; and {4} inadequacies in existing software technology

that prevent these technological advances from being fully exploited.

In the early sections of this work, the primary thesis is developed; it explicitly presents the properties that we require of a local network-based multiprocessor. The analysis and validation of this thesis leads to four major contributions. The first is a comparison of very high speed ring and broadcast networks when they are used with short packets. As part of this comparison, a new analytic model is presented, whose solution yields delay/throughput data for token rings.

The second major contribution is a new communication model for local computer networks whereby processes execute generalized remote references that cause operations to be performed by remote processes. This remote reference/remote operation model provides a taxonomy of primitives that are naturally useful in many applications and can be specially implemented to provide for high efficiency. Example communication primitives and techniques for their implementation are provided to show the utility of the model.

Following these discussions, we present experience with the implementation of one class of remote references. These references take about 150 microseconds or 50 average macroinstruction times to perform on Xerox Alto computers connected by a 2.97 megabit Ethernet. This experiment demonstrates the power of special-casing communication primitives and helps to validate the remote reference/remote operation model.

Finally, various implementation techniques are presented that can be used for a real communication system based upon the model. We discuss such topics as the efficient transmission of large blocks through the use of multiple small packets and the efficient implementation of stable storage.

[Spector 81b] Spector, Alfred Z.
Extending Local Network Interfaces to Provide More Efficient Interprocessor Communication Facilities.
In *Proceedings, Eighth Symposium on Operating Systems Principles*, pages 6-13. ACM, December, 1981.

Abstract

This paper describes extensions to local networking interfaces that allow high speed networks of processors to be used in certain multiprocessor applications. If network interfaces are augmented to permit more efficient message passing as well as direct memory access to other processors' memories, distributed applications requiring frequent interprocessor communication can be better supported. Resulting systems would have a hybrid architecture with characteristics of both shared memory and message passing systems, and could fully use the inherent reliability and extremely high bandwidth now provided by local networks.

The motivation and initial implementation plans for an experimental system to be contructed on Xerox Alto computers are discussed. Included in the prototype will be new machine instructions and Ethernet protocols that allow for both virtual shared memory and datagram operations. The resulting system will demonstrate that interprocess synchronization and communication can be performed much more efficiently by special purpose firmware than by current message passing mechanisms.

[Spier 73a] Spier, Michael J.
The Experimental Implementation of a Comprehensive Inter-Module Communication Facility.
In *Proceedings, 1973 Sagamore Computer Conference on Parallel Processing*, pages ?-? Syracuse University, August, 1973.

Abstract

In 1972, The Digital Equipment Corporation sponsored a limited-objective research project to investigate the properties of the new kernel/domain systems architecture, whose theoretical model was earlier developed by Spier. A companion paper reports on that project. The domain is a monitor (or supervisor, executive)-like local independent address space which may be mapped over a collection of (mostly) exclusive memory space partitions to provide a protected runtime environment. Similar to the classical monitor, control may be transferred into the domain through predesignated inter-domain entry points named gates. In a single monolithic monitor, but is distributed among a number of supervisory domains; of these, the most central and most critical supervisory domain is named kernel. The kernel is responsible for basic resource management only and is by definition devoid of any decision making code.

[Spier 73b] Spier, Michael J.
Process Communication Prerequisites or the IPC-Setup Revisited.
In *Proceedings, 1973 Sagamore Computer Conference on Parallel Processing*, pages 79-88. Syracuse University, August, 1973.

Abstract

A careful examination of any existing inter-process communication (IPC) mechanism invariably uncovers the underlying existence of a more fundamental IPC mechanism, which in turn is built on a yet more fundamental IPC mechanism...etc.

This study resolves this indefinite recursion of a self defining mechanism by proposing a certain causality, expressed in terms of a finite list of process communication prerequisites, and based on a non-mechanistic postulate which calls for an area of communication (or *mailbox*) that is by its very nature impervious to mutual interference by the communicating processes.

Given arbitrary processes for which these prerequisites hold, we may logically construct the "very first" *elementary IPC mechanism*, i.e., the one which is not dependent upon its own pre-existence. Such a mechanism is developed in this paper; it is capable of transmitting a single, one-way, one-bit message among processes.

It is suggested that the proposed causality, although arbitrary in many ways (and openly admitted as such) may serve as a convenient intellectual tool with which autonomous sequential processes may be observed and studied.

[Spratt 81] Spratt, E. Brian.
Operational Experiences with a Cambridge Ring Local Area Network in a University Environment.
In *Proceedings, IFIP Working Group 6.4, International Workshop on Local Networks*, pages 81-106. IBM, August, 1981.

Abstract

The University of Kent Computing Laboratory is responsible both for the central University Computing Service and for teaching and research in Computer Science. At the time of writing there is a total of some 1300 users, out of a total University population of 4100.

Interest in the Laboratory in Local Area Networks goes back to the early seventies.

[Stankovic 82] Stankovic, John A.
Software Communication Mechanisms: Procedure Calls Versus Messages.
Computer 15(4):19-25, April, 1982.

Abstract

Procedure calls and messages are two software communication techniques in wide use today. Whereas the semantics of the procedure call are well known, the newness and variety of message communication make it less understood.

Furthermore, the terms "procedure calls" and "messages" are often used in a general and imprecise manner, and therefore the differences between them tend to blur. This happens, for example, when the claim is made that messages can be programmed using procedure calls - a claim that is both true and, in fact, reflects what is often done in practice.

[Staunstrup 82] Staunstrup, Jorgen.
Message Passing Communication Versus Procedure Call Communication.
Software--Practice and Experience 12(?):223-234, 1982.

Abstract

Communication by message passing or by procedure calls in one of the key issues when discussing languages for multiprogramming. The two languages Platon and Concurrent Pascal represent the different approaches which are contrasted by presenting a few programs written in both languages.

[Stritter 81] Stritter, Edward P., Harry J. Saal and Leonard J. Shustek.
Local Networks of Personal Computers.
In *Digest of Papers, COMPCON 81 Spring*, pages 2-5. IEEE, 1981.

Abstract

The technologies of local computer networks and of personal computers are beginning to interact. Local networks enhance sharing and communication in a computer installation. Personal computers make significant

dedicated computer power available to the user at a cost that is little more than that of a terminal connected to a more traditional large system.

A commercially available local computer network of personal computers is described here. The system combines the advantages of personal computers (low cost per user, a computer on every desk, etc.) with those of local computer networks (access to shared resources, cost sharing of expensive peripherals, smooth system growth with constant compute power per user.)

Many new capabilities derive from local computer networks such as sharing of data, computer-to-computer communication, and intelligent server resources (shared high-speed printers, file systems, data-base backends, etc.) This paper discusses the network and internetwork configurations which make such capabilities possible.

[Stroet 80] Stroet, Jan.
An Alternative to the Communication Primitives in ADA.
ACM-SIGPLAN Notices 15(12):62-74, December, 1980.

Abstract
A critical look is taken at the ADA communication primitives by comparing them to the ITP (Input Tool Process) model, the model for process communiation developed at Nijmegen. The comparison is done by means of example solutions to several problems in both models. It is shown that by using features extracted from the ITP model, the communication facilities in ADA could be improved considerably with respect to orthogonality, clarity, flexibility and power.

[Stroustrup 79] Stroustrup, Bjarne.
An Inter-Module Communication System for a Distributed Computer System.
In *Proceedings, First International Conference on Distributed Computing Systems*, pages 412-418. IEEE, October, 1979.

Abstract
This paper outlines the design of an inter-module communication system suitable for a computer system consisting of many separate machines communicating via a local communication network. This inter-module communication system was designed to support the SIMOS operating system utilizing a number of such machines to provide services normally provided by a centralized system. Examples of how "server" machines can be used to run the SIMOS file system are presented together with data showing the effect of such usage on the overall system performance.

[Sunshine 76] Sunshine, Carl A.
Factors in Interprocess Communication Protocol Efficiency for Computer Networks.
In *Proceedings, National Computer Conference*, pages 571-576. AFIPS, 1976.

Abstract
This paper considers the efficiency of interprocess communication protocols for distributed processing environments such as computer networks. Previous research has emphasized system performance at lower levels, within the communication medium itself, while this work examines requirements and performance of protocols for communication between processes in the Host computers attached to the communication system. Efficiency primarily concerns throughput and delay achievable for communication between remote processes. Various aspects of protocol operation are analyzed, and protocol policies concerning retransmission, flow control, buffering, acknowledgment, and packet size emerge as the most important factors in determining efficiency. Several graphs showing quantitative performance results for representative situations are included.

[Sunshine 78] Sunshine, Carl A. and Yogen K. Dalal.
Connection Management in Transport Protocols.
Computer Networks 2(3):454-473, 1978.

Abstract
Transport protocols are designed to provide fully reliable communication between processes which must communicate over a less reliable medium such as a packet switching network (which may damage, lose, or duplicate packets, or deliver them out of order). This is typically accomplished by assigning a sequence number and checksum to each packet transmitted, and retransmitting any packets not positively acknowledged by the other side. The use of such mechanisms requires the maintenance of state information describing the progress of data exchange. The initialization and maintenance of this state information constitutes a connection between the two processes, provided by the transport protocol programs on each side of the connection. Since a connection

requires significant resources, it is desirable to maintain a connection only while processes are communicating. This requires mechanisms for opening a connection when needed, and for closing a connection after ensuring that all user data have been properly exchanged. These connection management procedures form the main subject of this paper. Mechanisms for establishing connections, terminating connections, recovering from crashes or failures of either side, and for resynchronizing a connection are presented. Connection management functions are intimately involved in protocol reliability, and if not designed properly may result in deadlocks or old data being erroneously delivered in place of current data. Some protocol modeling techniques useful in analyzing connection management are discussed, using verification of connection establishment as an example. The paper is based on experience with the Transmission Control Protocol (TCP), and examples throughout the paper are taken from TCP.

[Terada 80] Terada, M., J. Kashio, K. Yokota, Y. Hori and H. Fushimi.
A Network Operating System for High Speed Optical Fiber Loop Transmission System.
In *Proceedings, Fifth International Conference on Computer Communication: Increasing Benefits for Society*, pages 641-646. October, 1980.

Abstract

This paper describes a network operating system (NOS) developed for inhouse computer networks. The optical fiber loop system with a high transmission speed of 10 Mbits/sec connects some of the minicomputers and the many microcomputers. The NOS is designed to provide the following three features:

1. An improved interface between computer and transmission control equipment to achieve efficient data transfer.
2. Unified access interfaces to user programs with respect to the access to two kinds of resources, one being locally attached peripherals and the other remote devices. Centralized network system maintenance and operation functions for distributed mini/micro- computers, in order to obtain overall system efficiency and cost effectiveness.

[Test 79] Test, Jack A.
An Interprocess Communication Scheme for the Support of Cooperating Process Networks.
In *Proceedings, First International Conference on Distributed Computing Systems*, pages 405-411. IEEE, October, 1979.

Abstract

This paper describes an interprocess communication scheme for application in Distributed Operating System Environments. This scheme is based upon the notions of gates to processes and connections between gates. A gate, as conceived here, serves as a standard interface between the internal environment of a process and its external environment. A connection between gates allows a simplex information transfer between those gates. The proposed IPC primitives, when implemented as part of a distributed operating system kernel, provide some measure of built-in fault detection; enforce a capability-like scheme for gate access protection; and support multi-process dialogues.

[Thomas 76] Thomas, Robert H. and Stuart C. Schaffner.
MSG: The Interprocess Communication Facility for the National Software Works.
Technical Report 3483, Bolt Beranek and Newman Inc., December, 1976.

Abstract

The National Software Works (NSW) provides software implementers with a suitable environment for the development of programs. This environment consists of many software development tools (such as editors, compilers, and debuggers), running on a variety of computer systems, but accessible through a single access-granting, resource- allocating monitor with a single, uniform file system. By its very nature, the NSW consists of processes distributed over a number of computers connected by a communications network. These processes must communicate with one another in order to create a unified system. This paper describes the communication facility (named MSG) which was developed to provide interprocess communication for the implementation of the NSW. As we have noted, the communication network is currently the ARPANET. However, we have designed the MSG facility to be as independent as possible of the ARPANET implementation so that the concepts may carried over to implementation on other networks.

[Tilborg 80] Tilborg, Andre' M. van and Larry D. Wittie.
A Concurrent Pascal Operating System For a Network Computer.
In *Proceedings, COMPSAC 80*, pages 1-7. IEEE, October, 1980.

Abstract

A network computer (multi-micro-processor, modular computer) requires both a high-level control structure to bind the nodes into a cohesive computing device and a local operating system for each of the nodes to execute. This paper outlines the form of a hierarchical high-level control schema and describes a nodal operating system designed and built for the MICRONET network computer using Concurrent Pascal. The operating system consists of a packet switching subsystem which executes in the communications frontend processor of each node and a host processor operating system which manages local resources, interfaces to user terminals, and executes task forces. The host processor operating system customizes itself to the abilities of each node to some extent by not initializing some of its built-in Concurrent Pascal system components depending on the resources available at the host node. However, every node executes a process which supports the high-level control schema.

[Tilborg 82] Tilborg, Andre' M. van.
Packet Switching in the MICRONET Network Computer.
IEEE Transactions on Communications COM-30(6):1426-1433, June, 1982.

Abstract

Packet switching is a communication technology which has been used extensively in geographically distributed computer networks. It is also applicable to the communication subnetworks of compact multimicrocomputers known as network computers. This paper describes the use of the language Concurrent Pascal to build a packet-switching subsystem for the MICRONET network of DEC LSI-11 microcomputers. Examples of actual Concurrent Pascal source code taken from the system demonstrate the usefulness of high-level languages with abstract data types for complex communication software.

[Vervoort 80] Vervoort, W. A.
A Taxonomy of Interprocess Communication.
Technical Report, Twente University of Technology, 1980.

Abstract

A classification system for interprocess communication in Distributed Systems without shared variables has been developed based on three orthogonal choises with respect ot the measure of freedom of the processes in their communication. These three orthogonal axis construct a 3-D space of communication. Eleven example models of communication found in literature have been described and located in this space. Some points in the space remained unoccupied. Examples of the model closest to the origin (the most restricted-) and the most free communication model are given together with their basic concepts.

[Walden 72] Walden, David C.
A System for Interprocess Communication in a Resource Sharing Computer Network.
Communications of the ACM 15(4):221-230, April, 1972.

Abstract

A resource sharing computer network is defined to be a set of autonomous, independent computer systems, interconnected to permit each computer system to utilize all the resources of the other computer systems as much as it would normally call one of its own subroutines. This definition of a network and the desirability of such a network are expounded upon by Roberts and Wessler in [9]. Examples of resource sharing could include a program filing some data in the file system of another computer system, two programs in remote computer systems exchanging communications, or users simply utilizing programs of another computer system via their own.

[Walden 75] Walden, David C. and John M. McQuillan.
Some Consideration for a High Performance Message-Based Interprocess Communication System.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 45-54. ACM, March, 1975.

Abstract

We continue to be concerned with interprocess communications systems (such as those described in references 1, 2, and 3 and called "thin-wire" communications systems in reference 4) which are suitable for commmunication between processes that are not co-located in the same operating system but rather reside in different operating

systems on different computers connected by a computer communications network. Futher, the systems with which we are concerned are assumed to communicate using addressed messages (e.g., reference 5) which are multiplexed onto the logical communications channel between the source process and the destination process, rather than using such traditional methods as shared memory (an impossibility for distributed communicating processes) or dedicated physical communications channels between pairs of processes desiring to communicate (which is considered to be impossibly expensive).

[Walton 82] Walton, Robert L.
Rationale for a Queueable Object Distributed Interprocess Communication System.
IEEE Transactions on Communications COM-30(6):1417-1425, June, 1982.

Abstract

We consider the problem of designing an interprocess communication system usable as a base for writing real-time operating and applications systems in a distributed environment where processes may be connected by anything from shared virtual memory to radios. By requiring an interface that minimizes the code an application program must devote to communications, a facility of substantially higher level than basic message passing becomes necessary. This is largely a consequence of four major performance problems with interprocess communication in a distributed environment: system reliability, server congestion, throughput, and response time. We summarize these problems, and introduce an interprocess communication system based on two mechanisms: queueable objects and connectable objects. We briefly review our experience with a limited implementation of queueable objects.

[Watson 80] Richard W. Watson.
Distributed System Architecture Model.
[use the other one].

Abstract

The area of distributed systems is new and not well defined. The purpose of this chapter is to provide a conceptual framework for organizing the discussion of distributed system design goals, issues, and interrelationships, provide some common terminology to be used in the following chapters, and provide an overview of some common design issues. We refer to this framework or *reference architecture* as the *Distributed Systems Architecture Model* or simply as the *Model*. The remainder of the book elaborates the Model and presents alternative approaches to its realization. Besides serving as an organizing framework for the material of this book, we believe, the Model is useful in the design, organization, and analysis of a distributed system. The Model is shown in figure 2.1.

The Model contains three dimensions. The vertical dimension represents a distributed system as consisting of a set of logical layers. This book is primarily organized according to the categories on this axis. Each layer and sublayer has design and implementation issues unique to itself as well as a range of issues common among all the layers. These common issues are shown as a second dimension on the horizontal axis. The problems presented by each common issue and the appropriate solutions to it may differ in each layer. The third dimension, shown perpendicular to the page, concerns issues reflecting the global interaction of all parts of a destributed system on whole-system implementation and optimization. This dimension is poorly understood. It is shown here primarily as a reminder of its importance and the need for research to improve our understanding. Each of these dimensions is discussed in detail in the sections to follow.

[Wecker 80] Wecker, Stuart.
DNA: The Digital Network Architecture.
IEEE Transactions on Communications COM-28(4):510-526, April, 1980.

Abstract

Recognizing the need to share resources and distribute computing among systems, computer manufacturers have been designing network components and communication subsystems as a part of their hardware/software system offerings. A manufacturer's general purpose network structure must support a wide range of applications, topologies, and hardware configurations. The Digital Network Architecture, (DNA), the architectural model for the DECnet family of network implementations, has been designed to meet the specific requirements and to create a communications environment among the heterogeneous computers comprising Digital's systems.

This paper decribes the Digital Network Architecture, including an overview of its goals and structure, and details on the interfaces and functions within that structure. The protocols implementing the functions of DNA are described, including the motivations for the specific designs, alternatives and tradeoffs, and lessons learned from the implementations. The protocol descriptions include discussions of addressing, error control, flow control, synchronization, flexibility and performance. The paper concludes with examples of DECnet operation.

[Wettstein ??] Wettstein, H. and G. Merbeth.
The Concept of Asynchronization.
[unknown].

Abstract

Communication between parallel processes may take place in synchronous or asynchronous form. The former has widely been used in various concepts. In contrast, means for asynchronous process relations exist only in a few systems in rudimentary form. In this paper the concept of asynchronization is developed systematically. The underlying data structures as well as operations upon them are defined for various versions.

[Wittie 79] Wittie, Larry D.
A Distributed Operating System For a Reconfigurable Network Computer.
In *Proceedings, First International Conference on Distributed Computing Systems*, pages 669-677. IEEE, October, 1979.

Abstract

MICROS is the distributed operating system for the MICRONET network computer. MICRONET is a reconfigurable and extensible network of sixteen loosely-coupled LSI-11 microcomputer nodes connected packet-switching interfaces to pairs of high-speed shared communication buses. MICROS simultaneously supports many users, each running multicomputer parallel programs. MICROS is intended for control of network computers of up to ten thousand nodes.

Each network node is controlled by a private copy of the MICROS kernel processes written in Concurrent Pascal. Resource management tasks are distributed over the network in a control hierarchy. Management and user program tasks are Sequential Pascal programs dynamically loaded into the nodes. Whether in the same or different nodes, tasks communicate via a uniform message passing system. The MICROS command language allows spawning of groups of communicating tasks. Concurrent Pascal will eventually be provided for users writing parallel programs for MICRONET.

[Wulf 81] Wulf, William A., Roy Levin and Samuel P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw/Hill, New York, 1981.

Abstract

An operating system that encourages the use of cooperating sequential processes has a dual responsibility. On the one hand, it must provide protection mechanisms to insulate processes from one another so that erroneous or malicious behavior on the part of one cannot interfere with unrelated ones. On the other hand, it must also provide mechanisms for cooperation among the processes working on a common task. The last two chapters have dealt with some aspects of Hydra's response to the first of these responsibilities. In this chapter we shall deal with one aspect of the second.

Within the Hydra context, a wide range of interaction mechanisms are possible, from tightly coupled memory sharing to loosely coupled message communication. Moreover, the user is free to define application-specific mechanisms that lie anywhere along this spectrum. The Hydra Message System is a particular communication facility which we believe is convenient for many loosely coupled applications, and which can form the basis for many others.

[Xerox 81] Xerox Corporation.
Courier: The Remote Procedure Call Protocol.
Technical Report XSIS 038112, Xerox Corporation, December, 1981.

Abstract

One of the communication disciplines most frequently used by distributed system builders is that in which a request for service and its reply are exchanged by two system elements: a service provider and a service consumer. *Courier*, the Network System (NS) Remote Procedure Call Protocol, facilitates the construction of distributed systems by defining a single request/reply or transaction discipline for an open-ended set of higher-level application protocols. Courier standardizes the format of request and reply messages and the network representations for a family of data types from which request and reply parameters can be constructed.

Not all network communication is transaction-oriented. For example, the exchange of control information that typically precedes the transfer of a file between system elements might model naturally as a transaction. However, the transfer of the file's contents is more appropriately modeled as bulk data transfer.

Not all transaction-oriented communication is best accomplished using Courier. For example, the interrogation of a directory of network resources to locate a named resource might model naturally as a transaction. However,

satisfying the performance requirements for that operation might necessitate the use of datagrams, rather than virtual circuits (upon which Courier is based).

Other NS protocols--for example, the Sequenced Packet Protocol and the Internet Datagram Protocol [5]---support applications for which Courier is inappropriate.

[Zelkowitz 75] Zelkowitz, Marvin V.
A Proposal in Process Hierarchy and Network Communication.
In *Proceedings, SIGCOMM-SIGOPS Interface Workshop on Interprocess Communications*, pages 154-158. ACM, March, 1975.

Abstract

Network design was once a complex art that few understood but it is slowly becoming a science where many of the fundamental ideas are crystallizing into a set of basic axioms. The purpose of this note is to present one set of ideas and show how they can be developed into a reliable system. The system will be hierarchically structured and has a powerful protection mechanism that allows for reliable system operation.

[Zimmermann 81] Zimmermann, Hubert, Jean-Serge Banino, Alain Caristan, Marc Guillemont and Gerard Morisset.
Basic Concepts for the Support of Distributed Systems: The CHORUS Approach.
In *Proceedings, Second International Conference on Distributed Computing Systems*, pages 60-66. IEEE, April, 1981.

Abstract

Distribution brings completely new requirements for processing, synchronization, communication, protection, and engineering of distributed applications. The CHORUS architecture proposes a new approach to meet these requirements; the paper introduces a set of basic concepts and mechanisms essentially focused on run-time aspects of distributed systems.

A.3. Hardware Support for Operating Systems Architectures

[ACM 82] ACM.
Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems.
ACM, Palo Alto, California, 1982.
This Proceedings contains many papers on actual and proposed computer systems which incorporate some form of hardware support for operating systems.

[Ahuja.S 82] Ahuja, S.R. and Asthana, A.
A Multi-Microprocessor Architecture with Hardware Support for Communication and Scheduling.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 205-209. ACM, March, 1982.
A functionally partitioned multiprocessor system is described. A separate processor is provided to handle communication and scheduling functions, and there are special processors for handling I/O. The signalling and scheduling processor balances the load among the multiple execution units.

[Ames.S 83] Ames, S.R., Jr., Gasser, M., and Schell, R.R.
Security Kernel Design and Implementation: An Introduction.
IEEE Computer 16(7):14-22, July, 1983.
The security kernel is currently the most popular approach to designing very secure computer systems. It is defined as the hardware and software which provides a "reference monitor" abstraction such that every reference to information or change of authorization must pass through the monitor. The authors provide a good overview of the security kernel approach, and indicate that the hardware support required for efficient implementation of a security kernel is quite modest, and actually found on many commercially available systems.

[Anderson.G 75] Anderson, G.A. and Jensen, E.D.
Computer Interconnection Structures: Taxonomy, Characteristics, and Examples.
Computing Surveys 7(4):197-213, December, 1975.
Anderson and Jensen describe a taxonomy, or naming scheme, for systems of interconnected computers. This taxonomy is very useful for characterizing different system designs, and it provides a common context in which to compare them.

[Anderson.J 72] Anderson, J.A. and Lipovski, G.J.
A Cellular Processor for Task Assignments in Polymorphic, Multiprocessor Computers.
In *Proc. of Fall Joint Computer Conf.*, pages 703-708. AFIPS, December, 1972.
Anderson and Lipovski describe how an associative memory which does threshold matching can be used to quickly determine which job requests can be satisfied with current system resources.

[Anderson.L 75] Anderson, L.D.
Disciplined Software Development Utilizing a Hardware-Structured Executive.
In *Proc. of EASCON*, IEEE, September, 1975.
Anderson describes a system in which process descriptor segments are created, manipulated, and destroyed by hardware primitives, and process switching is carried out automatically as part of the basic machine cycle. Virtual address mapping is also aided by an associative memory containing the process ID along with the segment number. Using the process ID avoids having to switch memory mapping registers on process switches.

[Applewhi.H 79] Applewhite, H.L., Arnold, R.G., Gorman, T.J., Gouda, M.G., and Marks, C.P.
Modular Missile Borne Computer (MMBC) Software Structure and Implementation.
In *Proc. of 1st Int. Conf. on Distributed Computing Systems*, pages 725-735. IEEE, October, 1979.
MMBC is designed to support pipelined process structures, where each stage of the pipeline can have replicated processes implementing it. Such structures are felt to be very useful (and common) in high performance real time systems.

[Applewhi.H 80] Applewhite, H.L., Garg, R., Jensen, E.D., Northcutt, J.D., Sha, L., and Wendorf, J.W.
Distributed Computer Systems: Fiscal Year Interim Report to Rome Air Development Center, October 1980.
Carnegie-Mellon University, Computer Science Department, 1980.
This report contains the initial paper on ArchOS, discussing some of the issues in the design of an operating system for a distributed computer system.

[Arden.B 81] Arden, B.W. and Ginosar, R.
MP/C: A Multiprocessor / Computer Architecture.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 3-19. IEEE and ACM, May, 1981.
MP/C is a dynamically partitionable multiprocessor system. A fast FORK operation is supported by partitioning the shared bus such that one processor (and associated process) is active in each partition. Adjacent partitions can later be JOINed, leaving one processor active in the combined partition.

[Atkinson.T 75] Atkinson, T.D., Gagliardi, U.O., Raviola, G., and Schwenk, H.S., Jr.
Modern Central Processor Architecture.
Proc. of the IEEE 63(6):863-870, June, 1975.
The operating system mechanisms supported in hardware by the Honeywell Series 60 Level 64 are described. The Level 64 provides automatic queueing and priority dispatching of processes, semaphores with and without messages, and a segmented virtual memory system with hardware protection rings. All I/O is handled through semaphores.

[Bal.S 82] Bal, S., Kaminker, A., Lavi, Y., Menachem, A., and Soha, Z.
The NS16000 Family - Advances in Architecture and Hardware.
IEEE Computer 15(6):58-67, June, 1982.
The NS16032 MPU is a 32-bit microprocessor with low level support for semaphores, and state saving on process switches. The NS16082 MMU provides virtual address translation and memory protection.

[Balzer.R 73] Balzer, R.M.
An Overview of the ISPL Computer System Design.
Communications of the ACM 16(2):117-122, February, 1973.
Balzer stresses the advantages of concurrent design of the programming language, operating system, and machine architecture of a computing system. The ISPL computer system provides support in microcode for process scheduling, memory allocation, and a port mechanism for uniform communication with files, devices, and processes.

[Barton.G 82] Barton, G.C.
Sentry: A Novel Hardware Implementation of Classic Operating System Mechanisms.
In *Proc. of 9th Annual Symp. on Computer Architecture*, pages 140-147. IEEE and ACM, April, 1982.
The Sentry is a hardware memory protection mechanism. It monitors activity on the system bus and blocks those references which are not permitted in the active process.

[Bell.C 82] Bell, C.G., Newell, A., Reich, M., and Siewiorek, D.P.
The IBM System/360, System/370, 3030, and 4300: A Series of Planned Machines that Span a Wide Performance Range.
In Siewiorek, D.P., Bell, C.G., and Newell, A., editor, *Computer Structures: Principles and Examples*, pages 856-892. McGraw-Hill, 1982.
This article provides a good survey of the range of IBM System/360, System/370, and follow-on machines. The distinguishing characteristics and extra options for the various models are all briefly examined. A number of models are found to include various types and levels of support for operating system functions.

[Berenbau.A 82] Berenbaum, A.D., Condry, M.W., and Lu, P.M.
The Operating System and Language Support Features of the BELLMAC-32 Microprocessor.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 30-38. ACM, March, 1982.
The BELLMAC-32 is a 32-bit microprocessor which provides a number of mechanisms for making operating system implementation easier. It "understands" process control blocks, providing instructions for switching processes. I/O interrupts cause automatic process switches.

[Berg.R 71] Berg, R.O. and Thurber, K.J.
A Hardware Executive Control for the Advanced Avionic Digital Computer System.
In *Proc. of National Aerospace Electronics Conf.*, pages 206-213. IEEE, May, 1971.
The AADC is a real time, multiprocessor system containing a special hardware unit called Master Executive Control (MEC). The MEC provides all executive control for the system, handles interrupts, and does all scheduling on the basis of priority and importance criteria. The MEC uses an associative memory to aid in resource management by making searches for status information very fast.

[Berndt.H 76] Berndt, H.
Evolutionary Computer Architecture: The Unidata 7.000 Series.
Computer Architecture News 5(1):10-16, April, 1976.
In the Unidata 7.000 Series, process switching is aided by Store Status of Program (SSP) and Load Status of Program (LSP) instructions which save and restore the process state registers.

[Bernhard.R 81] Bernhard, R.
More Hardware Means Less Software.
IEEE Spectrum 18(12):30-37, December, 1981.
Bernhard provides a good, balanced introduction to the Reduced Instruction Set Computer (RISC) versus Complex Instruction Set Computer (CISC) controversy.

[Blaauw.G 64] Blaauw, G.A. and Brooks, F.P., Jr.
The Structure of System/360, Part I: Outline of the Logical Structure.
IBM Systems Journal 3(2):119-135, 1964.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 695-706.

[Boebert.W 77] Boebert, W.E., Bonneau, C.H., and Carnall, J.J.
Secure Computing.
In *Proc. of Symp. on Trends and Applications 1977: Computer Security and Integrity*, pages 49-63. IEEE and NBS, May, 1977.
The "Secure Communications Processor" (SCOMP) is discussed, both in terms of the underlying design issues and the actual implementation. SCOMP supports multilevel security through special purpose software running on a modified and enhanced Honeywell Level 6 minicomputer. A special hardware Security Protection Module (SPM) mediates all processor to memory, processor to device, and device to memory interactions.

[Boebert.W 78a] Boebert, W.E., Franta, W.R., Jensen, E.D., and Kain, R.Y.
Decentralized Executive Control in Distributed Computer Systems.
In *Proc. of COMPSAC '78*, pages 254-258. IEEE, November, 1978.
This paper discusses the issues and requirements involved in the design of the decentralized executive for a distributed computer system (HXDP).

[Boebert.W 78b] Boebert, W.E., Franta, W.R., Jensen, E.D., and Kain, R.Y.
Kernel Primitives of the HXDP Executive.
In *Proc. of COMPSAC '78*, pages 595-600. IEEE, November, 1978.
The HXDP Executive is primarily just a communication kernel. The structure of processes (virtual processors) and the communication mechanisms (ports) are described.

[Boehm.B 83] Boehm, B.W.
The Hardware / Software Cost Ratio: Is It a Myth?
IEEE Computer 16(3):78-80, March, 1983.
Boehm responds to Cragon's claim that the hardware / software cost ratio is a myth by pointing out that one must be careful about the situations in which it is applied. For the entire United States, the law seems to hold. However there are a number of more narrow contexts in which the law should not be applied.

[Brandwaj.A 79] Brandwajn, A., Hernandez, J.A., Joly, R., and Kruchten, Ph.
Overview of the ARCADE System.
In *Proc. of 6th Annual Symp. on Computer Architecture*, pages 42-49. IEEE and ACM, April, 1979.
ARCADE is a multiprocessor system with each terminal attached to its own "slow" processor, which handles most operating system tasks. Application processes are assigned dedicated "fast" processors and memory modules, selected from a pool of such components. Hardware implemented resource allocation lists are provided to the slow processors to aid in allocating the fast processors and memory modules among the application processes.

[Broadben.J 74] Broadbent, J.K. and Coulouris, G.F.
MEMBERS - A Microprogrammed Experimental Machine With a Basic Executive for Real-Time Systems.
SIGPLAN Notices 9(8):154-160, August, 1974.
I{Proc.ofACMSIGPLAN-SIGMICROInterfaceMeeting
.}

[Brown.G 77] Brown, G.E., Eckhouse, R.H., Jr., and Estabrook, J.
Operating System Enhancement Through Firmware.
In *Proc. of Micro10: 10th Annual Workshop on Microprogramming*, pages 119-133. IEEE and ACM, October, 1977.
The paper looks at the improvements possible through implementing parts of the operating system nucleus in microcode. Queue manipulation and semaphores are particular mechanisms that are investigated. A model of a simple timesharing system shows that a 70 percent reduction in nucleus execution time will result in about a 25 percent reduction in response time.

[Budzinsk.R 82] Budzinski, R.L., Linn, J., and Thatte, S.M.
A Restructurable Integrated Circuit for Implementing Programmable Digital Systems.
IEEE Computer 15(3):43-54, March, 1982.
The RIC chip contains four 16-bit processor slices which can be connected in various ways. One possibility is to use two of the processors in lockstep to form a 32-bit application processor while the other two processor slices are used for operating system and I/O processing.

[Burkhard.W 73] Burkhardt, W.H. and Randel, R.C.
Design of Operating Systems with Micro-Programmed Implementation.
Technical Report PIT-CS-BU-73-01, Univ. of Pittsburgh, Computer Science Dept., September, 1973.
Also available as NTIS Report PB-224-484.

[Buzen.J 73] Buzen, J.P. and Gagliardi, U.O.
The Evolution of Virtual Machine Architecture.
In *Proc. of National Computer Conf.*, pages 291-299. AFIPS, June, 1973.
Buzen and Gagliardi survey the hardware and software methods which have been employed to support virtual machines on existing "third generation" architectures.

[Case.R 78] Case, R.P. and Padegs, A.
Architecture of the IBM System/370.
Communications of the ACM 21(1):73-96, January, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 830-855.

[Cheriton.D 79] Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, February, 1979.
The primary concepts and facilities of the Thoth real time operating system are described. Process structuring of programs is emphasized. The communication mechanism only provides for synchronous sends of messages to the single mailbox associated with a receiving process.

[Clark.D 80] Clark, D.W. and Strecker, W.D.
Comments on "The Case for the Reduced Instruction Set Computer," by Patterson and Ditzel.
Computer Architecture News 8(6):34-38, October, 1980.
Clark and Strecker respond to the paper by Patterson and Ditzel, pointing out a number of weaknesses in the arguments which they gave in favor of reduced instruction set computers. Clark and Strecker believe that it will be very difficult to compare RISC and and CISC architectures without actually building a complete RISC system, including the operating system, and evaluating it over a wide spectrum of real applications.

[Colwell.R 83] Colwell, R.P., Hitchcock, C.Y., III, Jensen, E.D.
Peering Through the RISC/CISC Fog: An Outline of Research.
Computer Architecture News 11(1):44-50, March, 1983.
The authors propose two studies designed to shed more light on the current RISC/CISC debate. First they want to separate out the performance degradation caused by object orientation overhead, from degradation caused by complexity of the instruction set itself in machines such as the iAPX 432. The second study is to separate the performance gains due to multiple register set techniques, from those resulting from the reduced complexity of the instruction set itself in RISC machines.

[Copeland.G 82] Copeland, G.P.
What If Mass Storage Were Free?
IEEE Computer 15(7):27-35, July, 1982.
Copeland investigates the possible advantages of a nondeletion strategy for a mass storage system, including increased functionality through access to past states, and improved system performance through avoidance of garbage collection, reduced need for checkpoints, and reduced need for locking.

[Cragon.H 82] Cragon, H.G.
The Myth of the Hardware / Software Cost Ratio.
IEEE Computer 15(12):100-101, December, 1982.
Cragon questions the "folk law" which states that today software costs are two to four times the cost of hardware. He cites a number of studies in supporting his contention that the cost of software is high, but less than the cost of hardware.

[Dahlby.S 78] Dahlby, S.H., Henry, G.G., Reynolds, D.N., and Taylor, P.T.
System/38: A High Level Machine.
In *IBM System/38: Technical Developments*, pages 47-50. IBM GS80-0237, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 533-536.

[Dannenbe.R 79] Dannenberg, R.B.
An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages.
In *Proc. of 6th Annual Symp. on Computer Architecture*, pages 50-57. IEEE and ACM, April, 1979.
Dannenberg discusses a number of techniques for using many registers to hold the variables of a program. However, there is no discussion of the problems such large numbers of registers cause for process switching.

[DeBruijn.N 67] DeBruijn, N.G.
Additional Comments on a Problem in Concurrent Programming Control.
Communications of the ACM 10(3):137-138, March, 1967.
DeBruijn modifies Knuth's solution to the critical section mutual exclusion problem so that an individual process is guaranteed access to its critical section within N(N-1)/2 turns.

[DeMartin.M 76] DeMartinis, M., Lipovski, G.J., Su, S.Y.W, and Watson, J.K.
A Self Managing Secondary Memory System.
In *Proc. of 3rd Annual Symp. on Computer Architecture*, pages 186-194. IEEE and ACM, January, 1976.
The authors show how, by adding associative hardware to serial memory devices, the file system can become self managing in that no directories need be kept and garbage collection and storage allocation can be provided automatically.

[Denning.P 68] Denning, P.J.
The Working Set Model for Program Behavior.
Communications of the ACM 11(5):323-333, May, 1968.
This is the first paper discussing the working set model for memory management. The working set of a process is the set of pages referenced by that process in a given "window" of virtual time. A process will not be allowed to execute unless all of its working set can fit in main memory. In this way the load on the processor is automatically controlled, and thrashing is avoided. Denning discusses two implementations of the working set model, first assuming only that a "use bit" is associated with each page in memory, and second assuming that a timer is associated with each page.

[Denning.P 80a] Denning, P.J.
Why Not Innovations in Computer Architecture?
Computer Architecture News 8(2):4-7, April, 1980.
Denning laments the fact that proven techniques such as virtual storage management, among others, are not (properly) incorporated in most commercial architectures, in spite of convincing demonstrations of their value.

[Denning.P 80b] Denning, P.J. and Dennis, T.D.
On Minimizing Contention at Semaphores.
Computer Architecture News 8(2):12-19, April, 1980.
The use of tagged memory and microprogrammed operations are explored as means of keeping the holding times of semaphores and the process ready list manipulations to a minimum in multiprocessor systems.

[Denning.P 82] Denning, P.J.
Are Operating Systems Obsolete?
Communications of the ACM 25(4):225-227, April, 1982.
Denning argues that the principal concepts of operating systems can be grouped into five broad classes: Process Coordination, Virtual Memory, File System, Device Independence, and Job Control. Furthermore, these principal concepts will not become obsolete in the near future.

[Dijkstra.E 65] Dijkstra, E.W.
Solution of a Problem in Concurrent Programming Control.
Communications of the ACM 8(9):569, September, 1965.
Dijkstra shows how mutually exclusive access to the critical section in each of N concurrent, sequential processes can be ensured, assuming only that indivisible read and write operations on the primary memory are available.

[Dijkstra.E 68] Dijkstra, E.W.
The Structure of the "THE" - Multiprogramming System.
Communications of the ACM 11(5):341-346, May, 1968.
The THE operating system was structured as a hierarchy of nested abstract machines. The hierarchy was implemented as a series of layers of software, each extending the instruction set of the machines below it, and hiding the details of its internal structure from the levels above.

[Ditzel.D 80a] Ditzel, D.R. and Patterson, D.A.
Retrospective on High-Level Language Computer Architecture.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 97-104. IEEE and ACM, May, 1980.
Ditzel and Patterson argue for paying more attention to developing a High Level Language Computer System (HLLCS), rather than just a high level language computer. They list the attributes of a HLLCS, one of which is support for operating systems.

[Ditzel.D 80b] Ditzel, D.R. and Kwinn, W.A.
Reflections on a High Level Language Computer System or Parting Thoughts on the SYMBOL Project.
In *Proc. of Int. Workshop on High-Level Language Computer Architecture*, pages 80-87. Dept. of Computer Science, Univ. of Maryland, May, 1980.
Ditzel and Kwinn comment on various aspects of the SYMBOL System. The hardware implemented operating system was very successful from a performance and programming standpoint, but while software costs were reduced, overall costs were not.

[Ditzel.D 82] Ditzel, D.R. and McLellan, H.R.
Register Allocation for Free: The C Machine Stack Cache.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 48-56. ACM, March, 1982.
The stack cache mechanism improves the speed of subroutine calls and access to most operands. Unfortunately, process switching time is increased since the entire cache register file must be saved and restored.

[Eads.W 82] Eads, W.D., Walden, J.M., and Miller, E.L.
A Dual-Processor Desk-Top Computer: The HP 9845A.
In Siewiorek, D.P., Bell, C.G., and Newell, A., editor, *Computer Structures: Principles and Examples*, pages 508-532. McGraw-Hill, 1982.
The HP 9845A contains two main processors, a Language Processing Unit for interpreting BASIC programs, and a Peripheral Processing Unit for handling I/O and most management functions normally associated with an operating system.

[Eisenber.M 72] Eisenberg, M.A. and McGuire, M.R.
Further Comments on Dijkstra's Concurrent Programming Control Problem.
Communications of the ACM 15(11):999, November, 1972.
Eisenberg and McGuire improve upon DeBruijn's and Knuth's solutions to the critical section mutual exclusion problem so that an individual process is guaranteed access to its critical section within N-1 turns.

[Erwin.J 70] Erwin, J.D. and Jensen, E.D.
Interrupt Processing with Queued Content-Addressable Memories.
In *Proc. of Fall Joint Computer Conf.*, pages 621-627. AFIPS, November, 1970.
Erwin and Jensen describe the design of a special purpose Interrupt Processor (IP) which incorporates all of the functions associated with detecting, acknowledging, and scheduling interrupts on a priority basis. The IP is organized around a special unit called a queued content-addressable memory, which forms its primary storage and processing facility.

[Fabry.R 74] Fabry, R.S.
Capability Based Addressing.
Communications of the ACM 17(7):403-412, July, 1974.
Fabry provides a good overview of capability based addressing and protection mechanisms, their motivation, and their implementation.

[Fancott.T 77] Fancott, T. and Probst, W.G.
Software Distribution in a Microcomputer-Based Multiprocessor.
In *Proc. of 6th Texas Conf. on Computing Systems*, pages 4B.28-4B.34. IEEE and ACM, November, 1977.
Fancott and Probst suggest that OS modules could eventually be provided as standard chips and then interconnected with some form of bus. Message communication would be used among the functional modules. Suggested modules are device service routines, file management package, task scheduler, resource allocator, remote communications controller, and general processors for user tasks. Prior to the availability of such standard chips, microprocessors could be used.

[Farber.D 72] Farber, D.J., and Larson, K.C.
The Structure of a Distributed Computing System - Software.
In *Proc. of Symp. on Computer-Communications Networks and Teletraffic*, pages 539-545. Polytechnic Press, April, 1972.

[Flynn.M 72] Flynn, M.J. and Podvin, A.
Shared Resource Multiprocessing.
IEEE Computer 5(2):20-28, March/April, 1972.
Flynn and Podvin propose an extension to the hardware timeshared ALU approach as found in the peripheral processors of the CDC 6600. 32 skeleton processors, divided into 4 rings of 8 processors each, share multiple, high performance, pipelined execution units. A maximum performance of 500 MIPS is claimed to be possible.

[Ford.W 76] Ford, W.S. and Hamacher, V.C.
Hardware Support for Inter-Process Communication and Processor Sharing.
In *Proc. of 3rd Annual Symp. on Computer Architecture*, pages 113-118. IEEE and ACM, January, 1976.
Ford and Hamacher describe the hardware implementation of a simple single-word mailbox communication mechanism which can be used as a basis for more complex communication. All I/O is done through the mailbox mechanism. A hardware priority dispatcher, which can overlap with normal processing, provides fast process switches by having a separate register set for each process.

[Fraim.L 83] Fraim, L.J.
Scomp: A Solution to the Multilevel Security Problem.
IEEE Computer 16(7):26-34, July, 1983.
The Honeywell Secure Communications Processor (Scomp) is a commercially available minicomputer system supporting a multilevel security policy. The system is based on a security kernel, with special hardware, called the Security Protection Module, added to enhance the performance of the reference mediation operations.

[Freeman.M 78] Freeman, M., Jacobs, W.W., and Levy, L.S.
Perseus: An Operating System Machine.
In *Proc. of 3rd USA-Japan Computer Conf.*, pages 430-435. AFIPS and IPSJ, October, 1978.
Perseus consists of three main modules. The Supervisor receives user requests and sequences the actions to be performed. The Interface, which consists of a memory m ager, resource manager, dispatcher, action processor(s), and I/O control, does resource allocation and carries out actions. The Policy Module monitors system performance and adjusts paramenters and procedures to meet varying system loads.

[Gerrity.G 81] Gerrity, G.W.
On Processes and Interrupts.
Computer Architecture News 9(4):4-14, June, 1981.
Gerrity discusses hardware support for process queueing and scheduling. WAKE-UP and SLEEP operations are used for synchronization and communication. Interrupts are provided as WAKE-UP signals. Process switching is handled automatically and is aided by the use of "sticky bits", so that only modified registers are saved, and "undefined bits", so that only registers which are used are loaded.

[Gifford.D 77] Gifford, D.K.
Hardware Estimation of a Process' Primary Memory Requirements.
Communications of the ACM 20(9):655-663, September, 1977.
In the Honeywell 6180 processor supporting Multics, an associative table keeps the 16 most recently used page names in LRU order. By keeping track of the miss rate of this associative memory it is possible to estimate the working set size of a process, since the two should be proportional.

[Giloi.W 81] Giloi, W.K. and Behr, P.
An IPC Protocol and its Hardware Realization for a High-Speed Distributed Multicomputer System.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 481-493. IEEE and ACM, May, 1981.
Each node in the system has a separate Cooperation Handler processor for supporting message communication according to a producer and consumer type of protocol. The Cooperation Handler also provides some protection by controlling access to local objects from remote nodes. This, in cooperation with the Address Transformation and Memory Guard Unit provides protected, capability based access to the local memory of a node.

[Goldberg.R 73] Goldberg, R.P.
Architecture of Virtual Machines.
In *Proc. of National Computer Conf.*, pages 309-318. AFIPS, June, 1973.
Goldberg presents a model of recursive virtual machines as a compound mapping of process names into resource names, and virtual resource names into real resource names. He proposes a "hardware virtualizer" as the natural implementation of this model and suggests that a virtual machine with this support should enjoy performance comparable to the real machine.

[Goldberg.R 74] Goldberg, R.P.
A Survey of Virtual Machine Research.
IEEE Computer 7(6):34-45, June, 1974.
Goldberg surveys a variety of new architectures which are specifically designed to support virtual machines.

[Goldstei.B 75] Goldstein, B.C. and Scrutchin, T.W.
A Machine-Oriented Resource Management Architecture.
In *Proc. of 2nd Annual Symp. on Computer Architecture*, pages 214-219. IEEE and ACM, January, 1975.
An APL-like machine is described in which hardware supported locks or arbitrary software management functions can be associated with any object to control its use. The control function is invoked automatically whenever the object is referenced.

[Guillier.P 80] Guillier, P. and Slosberg, D.
An Architecture with Comprehensive Facilities of Inter-Process Synchronization and Communication.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 264-270. IEEE and ACM, May, 1980.
The hardware support for processes, semaphores, and messages in the Honeywell Series 60 Level 64 is described in some detail. The Level 64 provides automatic queueing and priority dispatching of processes. Semaphores with and without messages are supported and all I/O is handled through such semaphores.

[Halstead.R 80] Halstead, R.H.,Jr., and Ward, S.A.
The MuNet: A Scalable Decentralized Architecture for Parallel Computation.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 139-145. IEEE and ACM, May, 1980.

[Hatch.T 68] Hatch, T.F., Jr. and Geyer, J.B.
Hardware / Software Interaction on the Honeywell Model 8200.
In *Proc. of Fall Joint Computer Conf.*, pages 891-901. AFIPS, December, 1968.
The Model 8200 features hardware controlled "horizontal multiprogramming" whereby single instructions from each of up to 8 programs plus one master program (operating system) are executed in round robin sequence. The master program is given special privileges, including the ability to block execution of the other programs. A memory and peripheral device protection scheme based on locks and keys is supported.

[Hennessy.J 81] Hennessy, J., Jouppi, N., Baskett, F., and Gill, J.
MIPS: A VLSI Processor Architecture.
In Kung, H.T., Sproull, B., and Steel, G., editor, *VLSI Systems and Computations*, pages 337-346. Computer Science Press, 1981.
MIPS (Microprocessor without Interlocked Pipe Stages) is a high performance, reduced instruction set machine. The instruction set is essentially a compiler-driven encoding of the micromachine, so that little or no decoding is needed and the instructions correspond closely to microcode instructions. The processor is pipelined but provides no interlocks in hardware, relying instead on the compiler to arrange the code appropriately, inserting NO-OPs where necessary.

[Hennessy.J 82] Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J.
Hardware / Software Tradeoffs for Increased Performance.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 2-11. ACM, March, 1982.
The authors argue that the most effective system design methodology must make simultaneous tradeoffs across all three areas of hardware, software support, and systems support. The MIPS machine is used as an example.

[Hobson.R 81] Hobson, R.F.
Structured Machine Design: An Ongoing Experiment.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 37-55. IEEE and ACM, May, 1981.
The Structured Architecture Machine is a single user high level language computer system. It contains a separate Environment Control Unit which provides the traditional operating system functions such as task initiation, user command interpretation, peripheral communication, and so on.

[Hoffman.R 78] Hoffman, R.L. and Soltis, F.G.
Hardware Organization of the System/38.
In *IBM System/38: Technical Developments*, pages 19-21. IBM GS80-0237, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 544-546.

[Horton.F 74] Horton, F.R., Wagler, D.W., and Tallman, P.H.
Virtual Machine Assist: Performance and Architecture.
Technical Report TR 75.0006, IBM, April, 1974.
VM Assist is a set of microprograms for handling supervisor calls and 11 privileged instructions which were previously handled by VM/370 software. It provides a 75 percent reduction in supervisor state seconds and almost a 50 percent reduction in the elapsed time of batch throughput.

[Houdek.M 81] Houdek, M.E., Soltis, F.G., and Hoffman, R.L.
IBM System/38 Support for Capability-Based Addressing.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 341-348. IEEE and ACM, May, 1981.
The single level object store and capability-based addressing of the System/38 is described in some detail.

[Ichbiah.J 79] Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., and Wichmann, B.A.
Preliminary Ada Reference Manual and Rationale for the Design of the Ada Programming Language.
SIGPLAN Notices 14(6), June, 1979.
The original, preliminary definition of the Ada programming language, and an explanation of why some of its features were designed the way they were.

[ICOT 82] ICOT.
Outline of Research and Development Plans for Fifth Generation Computer Systems.
Technical Report, Institute for New Generation Computer Technology, Tokyo, Japan, May, 1982.
This report presents a good overview of the Japanese Fifth Generation Computer Systems Project. It contains slightly more detail than the paper by Treleaven and Lima dealing with the same topic.

[Intel 81a] Intel Corp.
iAPX 432 General Data Processor Architecture Reference Manual
Intel Corp., Santa Clara, CA, 1981.
The Intel iAPX 432 microprocessor could be characterized as an "operating system machine". It contains a powerful set of mechanisms in the areas of storage management, process scheduling, and interprocess communication. The 432 supports object-oriented systems.

[Intel 81b] Intel Corp.
iAPX 432 Interface Processor Architecture Reference Manual
Intel Corp., Santa Clara, CA, 1981.
The Intel iAPX 432 Interface Processor serves as an I/O channel. It extends the object and protection model of the 432 to the external interface, allowing processes to deal with external devices as objects. It also controls the access to main memory by external devices, enforcing the protection system.

[Intel 82] Intel Corp.
Software on Silicon: The iAPX 86/30 and 88/30.
Innovator 3(2):1-2, Winter, 1982.
A special chip, the 80130, contains the code for many basic operating system functions, and provides faster access than standard memory.

[Ishikawa.C 81] Ishikawa, C., Sakamura, K., and Maekawa, M.
Adaptation and Personalization of VLSI-Based Computer Architecture.
In *Proc. of Micro14: 14th Annual Workshop on Microprogramming*, pages 51-61. IEEE and ACM, October, 1981.
The authors discuss the advantages of monitoring and adapting a system to improve its performance. The primary adaptation technique is migration of frequently used, expensive functions into microcode. It is suggested that eventually the adaptation process will be done automatically.

[Jackson.P 83] Jackson, P.
Unix Variant Opens a Path to Managing Multiprocessor Systems.
Electronics 56(15):118-124, July, 1983.
The Convergent Technologies MegaFrame is a multiprocessor system which supports the Unix operating system. A set of Motorola 68010 based application processors handles all application related tasks including process and memory management. A separate set of Intel iAPX-186 based processors takes care of all file management, and another set of iAPX-186 based processors handles communications with peripheral devices.

[Jagannat.A 80] Jagannathan, A.
A Technique for the Architectural Implementation of Software Subsystems.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 236-244. IEEE and ACM, May, 1980.
Jagannathan shows how an operating system can be modelled using a "type extension" methodology. He argues that a given type need not be restricted to implementation in hardware or microcode or software simply on the basis of its position in the type hierarchy.

[Jenevein.R 81] Jenevein, R., Degroot, D., and Lipovski, G.J.
A Hardware Support Mechanism for Scheduling Resources in a Parallel Machine Environment.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 57-65. IEEE and ACM, May, 1981.
The authors discuss the hardware implementation of a simple scheduling algorithm for finding a processor that is "close" to a reference processor in a tree or SW-banyan interconnection network.

[Jensen.E 76] Jensen, E.D.
Distributed Processing in a Real-Time Environment.
In *Distributed Systems: Infotech State of the Art Report*, pages 303-318. Infotech, 1976.
Jensen describes the hardware support for message communication provided by the Modular Computer System, a forerunner of HXDP. In MCS the Global Bus Interface associated with each application processor provides hardware support for message queueing on output and receipt. It also handles most of the errors encountered in message communication.

[Jensen.E 78] Jensen, E.D.
The Honeywell Experimental Distributed Processor - An Overview.
IEEE Computer 11(1):28-38, January, 1978.
In HXDP each application processor has an associated Bus Interface Unit. The BIUs provide extensive support for message communication, especially in terms of error handling within the bus based communication system. Eight symbolic message destinations, each associated with an application process, are recognized by each BIU.

[Jensen.E 80] Jensen, E.D.
Distributed Computer Systems.
In Burks, S., editor, *Computer Science Research Review, 1979-1980*, pages 53-63. Carnegie-Mellon University, Computer Science Department, 1980.
Jensen discusses distributed computer systems, and in particular his model of decentralized resource management and control, and hardware / software relationships. He briefly outlines the Archons distributed computer system research project.

[Jensen.E 81a] Jensen, E.D.
Distributed Control.
In Lampson, B.W., Paul, M., and Siegert, H.J., editor, *Distributed Systems - Architecture and Implementation*, pages 175-190. Springer-Verlag, 1981.

[Jensen.E 81b] Jensen, E.D.
Hardware / Software Relationships in Distributed Computer Systems.
In Lampson, B.W., Paul, M., and Siegert, H.J., editor, *Distributed Systems - Architecture and Implementation*, pages 413-420. Springer-Verlag, 1981.
Jensen stresses the essential independence of two system design decisions which are often confused: layering and hardware versus software implementation. Layering involves deciding what functionality is performed at what layers in the system. Within each layer it is then necessary to decide how best to implement the functions of that layer.

[Jensen.K 74] Jensen, K. and Wirth, N.
Pascal User Manual and Report, 2nd ed.
Springer-Verlag, 1974.

[Johnsson.R 82] Johnsson, R.K. and Wick, J.D.
An Overview of the Mesa Processor Architecture.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 20-29. ACM, March, 1982.
Johnsson and Wick outline the main features of the Mesa processor. It supports monitors and condition variables, and provides event driven, rather than time sliced scheduling. All interrupts, exceptions, and communication with I/O devices use the process mechanism and condition variables.

[Jones.A 79] Jones, A.K., Chansler, R.J., Durham, I., Schwans, K., and Vegdahl, S.R.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proc. of 7th Symp. on Operating Systems Principles*, pages 117-127. ACM, December, 1979.
The microprogrammable Kmap processors of Cm* are used to support capability-based addressing of objects and a message communication mechanism. Operating system functions can be readily migrated to microcode since the Kmap is given "first refusal" on all system calls.

[Jones.A 82] Jones, A.K.
Private Communication, September 1982.
Jones indicated that there are a number of rules of thumb regarding operating system performance that circulate in the research community. All general purpose systems spend 30 to 50 percent of their time in the operating system. Operating system kernel entry and exit cost is 2 milliseconds, independent of the speed of the underlying machine. An I/O operation cannot be initiated in less than 10 milliseconds.

[Kamibaya.N 82] Kamibayashi, N., Ogawana, H., Nagayama, K., and Aiso, H.
Heart: An Operating System Nucleus Machine Implemented By Firmware.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 195-204. ACM, March, 1982.
Heart is an experiment to investigate the implementation of operating system kernel functions in microcode. It is expected that different virtual machines and operating systems could then be built on top of the universal and highly efficient primitives provided by Heart.

[Katsuki.D 78] Katsuki, D., Elsam, E.S., Mann, W.F., Roberts, E.S., Robinson, J.G., Skowronski, F.S., and Wolf, E.W.
Pluribus: An Operational Fault-Tolerant Multiprocessor.
Proc. of the IEEE 66(10):1146-1159, October, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 371-386.

[Kavi.K 82] Kavi, K., Belkhouche, B., Bullard, E., Delcambre, L., and Nemecek, S.
HLL Architectures: Pitfalls and Predilections.
In *Proc. of 9th Annual Symp. on Computer Architecture*, pages 18-23. IEEE and ACM, April, 1982.
The authors discuss some of the "myths" surrounding support for high-level languages. They suggest that support for operating systems and I/O is also very important since a machine can spend over half of its time executing operating system routines.

[Knowlton.K 65] Knowlton, K.C.
A Fast Storage Allocator.
Communications of the ACM 8(10):623-625, October, 1965.
This is the first description of the buddy system memory allocation algorithm.

[Knuth.D 66] Knuth, D.E.
Additional Comments on a Problem in Concurrent Programming Control.
Communications of the ACM 9(5):321-322, May, 1966.
Knuth points out that Dijkstra's solution to the critical section mutual exclusion problem can lead to starvation of individual processes. He provides a modification which guarantees access to the critical section by an individual process within 2^{N-1}-1 turns. He also points out that if indivisible queue manipulation operations were provided by the hardware, the solution would be much simpler and more efficient.

[Koplin.M 76] Koplin, M.R.
M138/M148 Performance Summary.
GUIDE Presentation, July 1976.

[Lamport.L 74] Lamport, L.
A New Solution of Dijkstra's Concurrent Programming Problem.
Communications of the ACM 17(8):453-455, August, 1974.
Lamport provides a new, simple solution to the critical section mutual exclusion problem which is more robust than previous solutions in that the system can continue to operate despite the failure of any individual component.

[Lampson.B 68] Lampson, B.W.
A Scheduling Philosophy for Multiprocessing Systems.
Communications of the ACM 11(5):347-360, May, 1968.
Lampson discusses processor scheduling and points out that a single hardware scheduler could be used in place of the interrupt system and software scheduler of usual systems. A well parameterized scheduler could carry out its functions quickly without being unduly restrictive, i.e. scheduling policies could be changed.

[Lampson.B 80] Lampson, B.W. and Pier, K.A.
A Processor for a High-Performance Personal Computer.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 146-160. IEEE and ACM, May, 1980.
The Dorado processor is capable of switching processes on every machine cycle. It uses a separate register set for each process in order to accomplish this. 16 tasks are supported, arranged in priority order, with task switching performed automatically in response to interrupts for higher priority tasks.

[Lampson.B 82a] Lampson, B.W.
Fast Procedure Calls.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 66-76. ACM, March, 1982.
Lampson argues that a processor's control transfer mechanism should handle a variety of applications such as procedure calls and returns, coroutine transfers, exceptions, and process switches in a uniform way. Furthermore, it should be very efficient for the common case of procedure call and return. The Mesa Processor's XFER primitive is based on the control transfer model presented in this paper.

[Lampson.B 82b] Lampson, B.W.
Private Communication, August 1982.
In summarizing the lessons learned from the BCC 500 Lampson stated, "The bottom line is that specialized processors are a fine idea, but won't make up for insufficient speed of the general processor, or for insufficient memory.".

[Landwehr.C 83] Landwehr, C.E.
The Best Available Technologies for Computer Security.
IEEE Computer 16(7):86-100, July, 1983.
Landwehr provides a good, concise overview of the work that has been done and is in progress in developing secure computer systems.

[Lee.W 74] Lee, W.K.
The Memory Management Function in a Multiprocessor Computer System - A Description of the BCC 500 Memory Manager.
Technical Report R-2, The Aloha System, Task II, Dept. of Electrical Engineering, Univ. of Hawaii, September, 1974.
Lee describes the BCC 500 Memory Management Processor in great detail. The memory manager is continuously active, monitoring the memory system and taking appropriate action more quickly and more often than is possible when time-sharing these functions on a CPU with other system and user tasks.

[Liskov.B 72] Liskov, B.H.
The Design of the Venus Operating System.
Communications of the ACM 15(3):144-149, March, 1972.
The Venus operating system consists of a combination of microcode and software. The microcode supports 16 virtual machines with priority scheduling. Semaphores are used for synchronization, communication, and I/O completion signalling.

[Maekawa.M 79] Maekawa, M., Yamazaki, I., Tanaka, A., Nakamura, A., and Ishida, K.
Experimental Polyprocessor System (EPOS) - Operating System.
In *Proc. of 6th Annual Symp. on Computer Architecture*, pages 196-201. IEEE and ACM, April, 1979.
EPOS is a functionally partitioned, multiprocessor system. Most of the operating system is implemented in microcode and the various functions can be reassigned dynamically to different processors.

[Maekawa.M 82] Maekawa, M., Sakamura, K., and Ishikawa, C.
Firmware Structure and Architectural Support for Monitors, Vertical Migration and User Microprogramming.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 185-194. ACM, March, 1982.
The microcode structure of a system (EPOS) is discussed in which the operating system kernel is implemented in microcode as a set of monitors. User microcode is supported by having a master (privileged) mode for system code and a slave mode for user code in the micromachine. The language interpreters are slave mode microcode.

[McCarthy.J 62] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I.
Lisp 1.5 Programmer's Manual.
MIT Press, 1962.

[McGehear.P 80] McGehearty, P.F.
Performance Evaluation of a Multiprocessor Under Interactive Workloads.
PhD thesis, Carnegie-Mellon University, Computer Science Department, August, 1980.
McGehearty measures and evaluates the performance of C.mmp/Hydra under a variety of synthetic, interactive workloads. One of the tools developed for this purpose was a Terminal Emulator, which is a separate processor that provides the synthetic, multiuser, interactive workloads based on stored scripts.

[Metcalfe.R 76] Metcalfe, R.M. and Boggs, D.R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM 19(7):395-404, July, 1976.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 429-438.

[Meyer.R 70] Meyer, R.A. and Seawright, L.H.
A Virtual Machine Time-Sharing System.
IBM Systems Journal 9(3):199-218, 1970.
Meyer and Seawright discuss in some detail the design and operation of Control Program-67 / Cambridge Monitor System (CP-67/CMS), one of the earliest virtual machine systems, and a forerunner of VM/370. CP-67 ran on the System/360 Model 67.

[Mitchell.J 79] Mitchell, J.G., Maybury, W., and Sweet, R.
Mesa Language Manual.
Technical Report CSL 79-3, Xerox Palo Alto Research Center, 1979.
This is the complete definition and reference manual for the Mesa programming language system.

[Morris.J 72] Morris, J.B.
Demand Paging Through Utilization of Working Sets on the MANIAC II.
Communications of the ACM 15(10):867-872, October, 1972.
Morris describes the design of a virtual memory system for the MANIAC II computer. Simple timer circuits for measuring elapsed process time since last access are added to the page frames of the system. These timers permit the cheap, direct measurement of intrinsic program working sets.

[Moto-oka.T 83] Moto-oka, T.
Overview to the Fifth Generation Computer System Project.
In *Proc. of 10th Annual Symp. on Computer Architecture*, pages 417-422. IEEE and ACM, June, 1983.
Moto-oka provides a nice, brief overview of the Japanese Fifth Generation Computer System Project, its goals and approaches.

[Muftic.S 77] Muftic, S. and Liu, M.T.
The Design of a Secure Computer System.
In *Proc. of Symp. on Trends and Applications 1977: Computer Security and Integrity*, pages 64-70. IEEE and NBS, May, 1977.
Muftic and Liu describe the design of a secure computer system in which special hardware devices for encoding and decoding data are added to all user terminals, and a Security Control Device mediates all CPU accesses to main memory. All data in main memory and in files can be stored in encoded form.

[Myers.G 80a] Myers, G.J. and Buckingham, B.R.S.
A Hardware Implementation of Capability-Based Addressing.
Computer Architecture News 8(6):12-24, October, 1980.
Similar material appears in Chapter 4 of Myers, *Advances in Computer Architecture*, Wiley, 1982.

[Myers.G 80b] Myers, G.J.
SWARD - A Software-Oriented Architecture.
In *Proc. of Int. Workshop on High-Level Language Computer Architecture*, pages 163-168. Dept. of Computer Science, Univ. of Maryland, May, 1980.
A more detailed discussion of SWARD is contained in Myers, *Advances in Computer Architecture*, Wiley, 1982.

[Myers.G 82] Myers, G.J.
Advances in Computer Architecture, Second Edition.
John Wiley & Sons, 1982.
Myers criticizes the conventional von Neumann architecture for leaving a large "semantic gap" between it and the concepts of modern high level languages and operating systems. He outlines various ways that this gap can be narrowed and discusses at great length a number of illustrative systems, including SYMBOL, SWARD, and the iAPX 432.

[Namjoo.M 82] Namjoo, M. and McCluskey, E.J.
Watchdog Processors and Capability Checking.
In *Proc. of 12th Annual Symp. on Fault-Tolerant Computing*, pages 245-248. IEEE, June, 1982.
Namjoo and McCluskey describe the use of a watchdog processor which monitors all activity on the processor to memory bus. The watchdog processor contains tables indicating the ways in which each object is permitted to access other objects. Any invalid accesses which are detected cause an error signal to be sent to the CPU.

[Nelson.B 81] Nelson, B.J.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, Computer Science Department, May, 1981.
One of Nelson's "performance lessons" in implementing remote procedure calls is to use microcode for exceptional performance. The physical transport time remains the same, but the enormous protocol overhead is drastically reduced.

[Nissen.S 73] Nissen, S.M. and Wallach, S.J.
The All Applications Digital Computer.
In *Proc. of Symp. on High-Level-Language Computer Architecture*, pages 43-51. IEEE and ACM, November, 1973.
AADC is a modular computer system in which the Data Processing Elements are specially designed to support the APL programming language. Many of the APL operators are directly supported in hardware. AADC also has extended hardware support for virtual memory management. Fifteen different page replacement algorithms are directly supported in hardware, and the choice of which algorithm to use is under program control.

[Organick.E 72] Organick, E.I.
The Multics System: An Examination of Its Structure.
MIT Press, 1972.
The structure of the Multics operating system is described in considerable detail.

[Organick.E 73] Organick, E.I.
Computer System Organization - The B5700/B6700 Series.
Academic Press, 1973.
The B5700/B6700 Series, like the B1700, provides microcode support for high-level languages and some operating system functions. There are hardware / microcode facilities for handling tasking, communication, and synchronization.

[Ousterho.J 80] Ousterhout, J.K., Scelza, D.A., and Sindhu, P.S.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.
In Medusa, microcode in the Kmap processors of Cm* provide the interprocess communication mechanism by supporting operations upon message pipes. Semaphores, indivisible increment, remote memory access, object descriptor manipulation, and fast block memory transfers are other facilities provided by microcode.

[Patterso.D 80] Patterson, D.A. and Ditzel, D.R.
The Case for the Reduced Instruction Set Computer.
Computer Architecture News 8(6):25-33, October, 1980.
Patterson and Ditzel look at the reasons behind the current preponderance of complex instruction set machines and find that the reasons are generally not very convincing. They also point out that CISCs have generally had a number of problems associated with them, such as increased design time and design errors. They suggest that a reduced instruction set approach would better serve the goal of supporting a high-level language computer sytem.

[Patterso.D 81] Patterson, D.A. and Sequin, C.H.
RISC I: A Reduced Instruction Set VLSI Computer.
In *Proc. of 8th Annual Symp. on Computer Architecture*, pages 443-457. IEEE and ACM, May, 1981.
RISC I has a simple instruction set so that almost all instructions execute in one machine cycle, essentially at microengine speed. A multiple overlapping register set scheme is used to allow very fast procedure call and return. However, the many registers would make process switching very slow.

[Pollack.F 82] Pollack, F.J., Cox, G.W., Hammerstrom, D.W., Kahn, K.C., Lai, K.K., and Rattner, J.R.
Supporting Ada Memory Management in the iAPX-432.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 117-131. ACM, March, 1982.
The Intel iAPX 432 capability based object addressing scheme is described in considerable detail. Stack, global heap, and local heap allocation of objects are all provided in hardware.

[Popek.G 75] Popek, G.J. and Kline, C.S.
The PDP-11 Virtual Machine Architecture: A Case Study.
In *Proc. of 5th Symp. on Operating Systems Principles*, pages 97-105. ACM, November, 1975.
Popek and Kline discuss the architectural changes needed to support a virtual machine system on a PDP-11/45. Ten sensitive instructions were modified to trap when executed in non-privileged mode and a performance enhancement unit was added to interpret most of a virtual machine's references to its upper 4K of memory (its I/O and status registers).

[Radin.G 82] Radin, G.
The 801 Minicomputer.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 39-47. ACM, March, 1982.
The 801 is a reduced instruction set machine. Radin emphasizes that the goal of such machines is to provide a simple, powerful instruction set that can be executed at about the speed of microcode. In this way, the operating system, as well as all applications, are essentially implemented in "microcode". A very powerful compiler technology is an essential part of a reduced instruction set computer, allowing all programming to be done in a high level language.

[Rashid.R 81] Rashid, R.F. and Robertson, G.G.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proc. of 8th Symp. on Operating Systems Principles*, pages 64-75. ACM, December, 1981.
In Accent, process management, interprocess communication, and virtual memory management are all supported in microcode. Process switching is aided by including the process ID as part of the virtual address so that no address mapping registers need be switched.

[Rattner.J 80] Rattner, J. and Cox, G.
Object-Based Computer Architecture.
Computer Architecture News 8(6):4-11, October, 1980.
Rattner and Cox discuss the object based computer architecture of the Intel iAPX 432. The function migration to hardware was done selectively to the best advantage from the standpoint of speed, space, and flexibility. Care was taken to keep resource management policies in software and put resource management mechanisms in hardware.

[Reghbati.H 78] Reghbati, H.K. and Hamacher, V.C.
Hardware Support for Concurrent Programming in Loosely Coupled Multiprocessors.
In *Proc. of 5th Annual Symp. on Computer Architecture*, pages 195-201. IEEE and ACM, April, 1978.
This paper expands on the work of Ford and Hamacher concerning hardware support for processes and single word mailboxes. It extends the idea to handle scheduled waits in monitors. A centralized process status table. rather than one per processor as before, is needed to support this type of global scheduling in a loosely coupled system.

[Richards.H 75] Richards, H., Jr. and Oldehoeft, A.E.
Hardware-Software Interactions in SYMBOL-2R's Operating System.
In *Proc. of 2nd Annual Symp. on Computer Architecture*, pages 113-118. IEEE and ACM, January, 1975.
SYMBOL supports 32 virtual processors, one per user with one reserved for operating system software. The hardware component of the OS, the System Supervisor, is a dedicated processor responsible for scheduling and paging. It invokes the OS software for other functions. The hardware scheduling algorithms permit software setting of various parameters, and the hardware page replacement algorithm takes into account processing mode, queue position, and type of data in order to make the best choice.

[Ritchie.D 74] Ritchie, D.M. and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.
This is the original paper describing the philosophy, design, and features of UNIX.

[Rosen.S 68] Rosen, S.
Hardware Design Reflecting Software Requirements.
In *Proc. of Fall Joint Computer Conf.*, pages 1443-1449. AFIPS, December, 1968.
Rosen briefly surveys the state of hardware support for the user visible "extended machine", as it stood in 1968. He suggests that interrupt handling, dynamic storage allocation, job management, compilation, and debugging aids are important areas in which hardware support could be beneficial.

[Rowan.J 75] Rowan, J.H., Smith, D.A., and Swensen, M.D.
Toward the Design of a Network Manager for a Distributed Computer Network.
In Feng, T., editor, *Parallel Processing: Proc. of Sagamore Computer Conf.; August 20-23, 1974*, pages 148-166. Springer-Verlag, 1975.
The authors describe a system in which a special purpose processor, called the Network Manager, interfaces to a number of functional nodes over one or more shared buses. The Network Manager provides a simple interprocessor message communication facility, and basic priority and deadline scheduling services.

[Ruggiero.M 80] Ruggiero, M.D. and Zaky, S.G.
A Microprocessor-Based Virtual Memory System.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 228-235. IEEE and ACM, May, 1980.
Ruggiero and Zaky describe a microprocessor system in which the virtual memory management is handled entirely by a separate microprocessor. In the current implementation the host processor is expected to simply wait while the page fault is handled. However there are a number of potential advantages of concurrent execution between the host and virtual memory processors. Paging out can be done ahead of time and more elaborate algorithms can be used at no extra cost.

[Rushby.J 83] Rushby, J. and Randell, B.
A Distributed Secure System.
IEEE Computer 16(7):55-67, July, 1983.
Rushby and Randell describe a proposed secure distributed system in which standard, untrustworthy Unix systems are connected to a shared local area network through special hardware units called Trustworthy Network Interface Units (TNIUs). Individual Unix systems are assigned to separate security classes, and the TNIUs primarily use encryption techniques to enforce multilevel security rules on the transmission of information between the systems.

[Saltzer.J 81] Saltzer, J.H., Reed, D.P., and Clark, D.D.
End-to-End Arguments in System Design.
In *Proc. of 2nd Int. Conf. on Distributed Computing Systems*, pages 509-512. IEEE, April, 1981.
The authors argue that system designers must think very carefully about the placement of functions in a layered system. Certain functions usually placed at low levels of the system are often redundant or of little value. In the context of message communication systems, the end-to-end argument basically states that if a function cannot be handled without the specialized knowledge and help of the application standing at both ends of the communication system, then the lower levels should not strain very hard to provide the function. At best they can enhance performance somewhat, but the application will still have to handle the function itself.

[Schroede.M 72] Schroeder, M.D. and Saltzer, J.H.
A Hardware Architecture for Implementing Protection Rings.
Communications of the ACM 15(3):157-170, March, 1972.
Rings of protection in Multics were originally supported by software. This paper suggests a hardware implementation of this mechanism so that most cross-ring CALL/RETURN operations take the same amount of time as regular CALL/RETURN.

[Schroede.S 73] Schroeder, S.C. and Vaughn, L.E.
A High Order Language Optimal Execution Processor: Fast Intent Recognition System (FIRST).
In *Proc. of Symp. on High-Level-Language Computer Architecture*, pages 109-116. IEEE and ACM, November, 1973.
In FIRST, Satellite Processing Units handle I/O and preliminary scheduling. The master processing unit consists of multiple machines for compiling and executing APL programs, and a separate processor for handling operating system functions such as job scheduling, resource allocation, library maintenance, diagnostics, and system error procedures. Communication among processors is through shared memory.

[SDS 68] Scientific Data Systems.
SDS Sigma 7 Computer Reference Manual
1968.
The Sigma 7 has 32 register sets where each register set can hold the state of a different process. As a result, fast process switching is possible by simply changing the active register set.

[Singer 73] Singer Business Machines.
System [Ten] Summary Manual
1973.
The Singer System Ten has a simple round-robin time-slicing supervisor implemented in hardware. Memory partition sizes are fixed at installation time and can be changed later.

[Sites.R 79] Sites, R.L.
How to Use 1000 Registers.
In *Proc. of CalTech Conf. on VLSI*, pages 527-532. CalTech Computer Science Dept., January, 1979.
Sites introduces the idea of using cached multiple register sets with a "dribble-back" saving technique and a prefetch restoring technique. Such a design will improve the speed of procedure call and return, but the many registers make process switching very slow. Sites suggests multiple such caches to improve process switching time.

[Smith.D 79] Smith, D.C.P. and Smith, J.M.
Relational Data Base Machines.
IEEE Computer 12(3):28-38, March, 1979.
Smith and Smith survey a variety of hardware support techniques for databases organized according to the relational model of data.

[Smith.W 71] Smith, W.R., Rice, R., Chesley, G.D., Laliotis, T.A., Lundstrom, S.F., Calhoun, M.A., Gerould, L.D., and Cook, T.G.
SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software.
In *Proc. of Spring Joint Computer Conf.*, pages 601-616. AFIPS, May, 1971.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 489-502.

[Sockut.G 75] Sockut, G.H.
Firmware / Hardware Support for Operating Systems: Principles and Selected History.
Technical Report TR 22-75, Harvard University, Center for Research in Computing Technology, October, 1975.
Sockut lists five proposed criteria for determining which operating system functions are the best candidates for hardware implementation. A selected history of the area is then presented, with very brief descriptions of a number of interesting systems and research efforts.

[Solomon.M 79] Solomon, M.H. and Finkel, R.A.
The Roscoe Distributed Operating System.
In *Proc. of 7th Symp. on Operating Systems Principles*, pages 108-114. ACM, December, 1979.

[Spector.A 82] Spector, A.Z.
Performing Remote Operations Efficiently on a Local Computer Network.
Communications of the ACM 25(4):246-260, April, 1982.
Spector describes a remote reference / remote operation communication model that can serve as the basis for a highly efficient communication subsystem. He stresses that efficient implementations may require the use of microcode or specialized hardware.

[Steel.R 77] Steel, R.
Another General Purpose Computer Architecture.
Computer Architecture News 5(8):5-11, April, 1977.
Steel describes an architecture in which a separate System Management Processor handles all process scheduling, management, and interprocess communication. One or more general purpose processors are permitted, and all I/O is handled by processes running on one or more I/O processors. Procedure activation record allocation and deallocation using overlapped records is handled automatically.

[Stockenb.J 73] Stockenberg, J.E., Anagnostopoulos, P.C., Johnson, R.E., Munck, R.G., Stabler, G.M., and Van Dam, A.
Operating System Design Considerations for Microprogrammed Mini-Computer Satellite Systems.
In *Proc. of National Computer Conf.*, pages 555-562. AFIPS, June, 1973.
This paper describes the structure of the operating system for the Brown University Graphics System (BUGS). There are 3 levels to the system with Level 0, the "hardware", simulated by a combination of microcode and software. Level 0 allows experiments with hardware versus software tradeoffs. It provides storage management, extended I/O, priority dispatcher, WAIT/POST facility, and extended interrupt generation and task creation.

[Stockenb.J 78] Stockenberg, J.E. and Van Dam, A.
Vertical Migration for Performance Enhancement in Layered Hardware / Firmware / Software Systems.
IEEE Computer 11(5):35-50, May, 1978.
General performance improvements can be achieved by avoiding the prologue and epilogue overheads associated with functions at higher levels. The authors describe a semi-automated methodology for determining which functions can most profitably be migrated downward. Although individual functions can be speeded up by a factor of 10 by migrating them to microcode, factors of 2 improvement are possible for applications which use the functions fairly heavily.

[Stonebra.M 81] Stonebraker, M.
Operating System Support for Database Management.
Communications of the ACM 24(7):412-418, July, 1981.
Stonebraker discusses various operating system functions and their usefulness in supporting database management systems (DBMS). Often these operating system facilities are found to be inadequate and must be provided anew by the DBMS. This situation reminds one somewhat of "the end-to-end argument" of Saltzer, et al.

[Strecker.W 78] Strecker, W.D.
VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family.
In *Proc. of National Computer Conf.*, pages 967-980. AFIPS, June, 1978.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 716-729.

[Stritter.E 79] Stritter, E. and Gunter, T.
A Microprocessor Architecture for a Changing World: The Motorola 68000.
IEEE Computer 12(2):43-52, February, 1979.
The 68000 has a relatively complex, orthogonal instruction set with many addressing modes, functions to aid procedure entry and exit, and functions to save and restore multiple registers.

[Su.S 79] Su, S.Y.W.
Cellular-Logic Devices: Concepts and Applications.
IEEE Computer 12(3):11-25, March, 1979.
Su surveys the use of cellular-logic devices to support database systems. In such devices a processing element is used for each circular memory element, and the concurrent processing of these elements allows fast data search and manipulation.

[Swan.R 77] Swan, R.J., Fuller, S.H., and Siewiorek, D.P.
Cm*: A Modular, Multi-Microprocessor.
In *Proc. of National Computer Conf.*, pages 637-644. AFIPS, 1977.
The microprogrammable Kmap processors in Cm* are intended to provide remote memory access for the various computer modules. However, they can also be used to implement many operating system functions, especially the interprocess communication facility.

[Thacker.C 82] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R.
Alto: A Personal Computer.
In Siewiorek, D.P., Bell, C.G., and Newell, A., editor, *Computer Structures: Principles and Examples*, pages 549-572. McGraw-Hill, 1982.
The Alto supports 16 tasks, each with a different priority level. Task switching to the highest priority ready task is performed semiautomatically in response to the TASK command. The various device controllers are quite intelligent, having the full power of the main micromachine available to them.

[Thornton.J 64] Thornton, J.E.
Parallel Operation in the Control Data 6600.
In *Proc. of Fall Joint Computer Conf., Pt. 2*, pages 33-40. AFIPS, 1964.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 730-736.

[Thurber.K 81] Thurber, K.J.
Hardware Issues.
In Lampson, B.W., Paul, M., and Siegert, H.J., editor, *Distributed Systems - Architecture and Implementation*, pages 377-412. Springer-Verlag, 1981.
Thurber briefly discusses the design of an architecture containing a system control unit which is separate from the application processor and provides the operating system kernel. The SCU provides process management, including synchronization (semaphores) and communication, and handles all I/O. It includes special state switch hardware to facilitate rapid application process switching.

[Tokoro.M 80] Tokoro, M., Tamaru, K., Mizuno, M., and Hori, M.
A High Level Multi-Lingual Multiprocessor KMP/II.
In *Proc. of 7th Annual Symp. on Computer Architecture*, pages 325-333. IEEE and ACM, May, 1980.
In KMP/II, an operating system processor and an I/O processor are statically assigned. The I/O processor schedules its own I/O processes, which include the file system processes. The OS processor allocates language emulation microcode among the application processors, provides interprocess communication facilities and scheduling of system and user processes, and handles all supervisor calls.

[Traiger.I 82] Traiger, I.L.
Virtual Memory Management for Database Systems.
Operating Systems Review 16(4):26-48, October, 1982.
Traiger discusses DBMS buffer management by describing two schemes, shadow paging and write ahead log, which can be used to ensure proper recovery from crashes. He then discusses the extensions necessary for a generalized virtual memory manager to be able to handle (most of) the operations now handled by the DBMS buffer manager. This would permit the mapping of files into virtual memory while maintaining recovery capabilities.

[Treleave.P 82] Treleaven, P.C. and Lima, I.G.
Japan's Fifth-Generation Computer Systems.
IEEE Computer 15(8):79-88, August, 1982.
Treleaven and Lima provide a good overview of the Japanese Fifth Generation Computer Systems Project. This is a very ambitious project aimed at developing "knowledge-information processing systems based on innovative theories and technologies that can offer the advanced functions expected to be required in the 1990's, overcoming the technical limitations inherent in conventional computers." The combined software and hardware of a fifth generation computer system is to provide three basic functions: the intelligent interface, knowledge-base management, and problem-solving and inference functions. There will be substantial hardware support for each of these functions.

[VanDeSne.J 79] Van de Snepscheut, J.L.A. and Slavenburg, G.A.
Introducing the Notion of Processes to Hardware.
Computer Architecture News 7(7):13-23, April, 1979.
By having multiple register sets and a queue of ready process IDs a very fast hardware process switching mechanism is possible, perhaps even a switch every instruction cycle. Hardware provided semaphore operations can be very fast by doing parallel operations on process control blocks maintained in special hardware cells. All I/O is assumed to use semaphores, so no separate interrupt structure is necessary.

[VonPuttk.E 75] Von Puttkamer, E.
A Simple Hardware Buddy System Memory Allocator.
IEEE Transactions on Computers C-24(10):953-957, October, 1975.
Von Puttkamer shows how the buddy system algorithm for memory allocation can be implemented very efficiently, and quite simply, using special hardware. Shift registers are used to maintain the binary tree which represents the current state of the memory system.

[Wall.C 74] Wall, C.F.
Design Features of the BCC 500 CPU.
Technical Report R-1, The Aloha System, Task II, Dept. of Electrical Engineering, Univ. of Hawaii, January, 1974.
The BCC 500 is a functionally partitioned multiprocessor designed to support a large number of time sharing users. There are 2 independent CPUs for running user programs, a process scheduling processor, a memory management processor, and a terminal handling processor. This report concentrates on the CPU architecture.

[Ward.S 79] Ward, S.A.
TRIX: A Network-Oriented Operating System.
Technical Report, MIT, December, 1979.

[Watson.W 72] Watson, W.J.
The TI ASC - A Highly Modular and Flexible Super Computer Architecture.
In *Proc. of Fall Joint Computer Conf.*, pages 221-228. AFIPS, December, 1972.
Reprinted in Siewiorek et al., *Computer Structures: Principles and Examples*, McGraw-Hill, 1982, pp. 753-762.

[Wegner.P 80] Wegner, P.
Programming with Ada: An Introduction by Means of Graduated Examples.
Prentice-Hall, 1980.

[Wendorf.J 83] Wendorf, J.W.
Hardware Support for Operating System Architectures.
In preparation, Computer Science Department, Carnegie-Mellon University.

[Wendorf.R 82] Wendorf, R.G.
Decentralized Resource Management.
In *Definition of Distributed Operating System Concepts and Techniques: NOSC Contract N66001-81-C-0484 First Quarterly Progress Report, 20 January 1982.* Carnegie-Mellon University, Computer Science Department, 1982.

[Werkheis.A 70] Werkheiser, A.H.
Microprogrammed Operating Systems.
In *Preprints of 3rd Annual Workshop on Microprogramming*, pages III.C.1-III.C.16. IEEE and ACM, October, 1970.
Werkheiser divides operating system functions into 3 levels: miniprimitives (data structure manipulation and searching, subroutine linkage, etc.); midiprimitives (IPC, semaphores, memory management, file allocation, scheduling, etc.); and maxiprimitives (compilers, loaders, etc.). He suggests which primitives for each level are most appropriate for microprogrammed implementation, based on his suggested criteria for making such implementation decisions.

[Wilkes.M 82] Wilkes, M.V.
Hardware Support for Memory Protection: Capability Implementations.
In *Proc. of Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 107-116. ACM, March, 1982.
Wilkes discusses the present state of hardware supported capability systems and makes some suggestions on how to reduce their complexity. He believes that it is still an open question whether the capability approach can reduce the cost of supporting small and frequently changing domains of protection to reasonable proportions.

[Wilner.W 72] Wilner, W.T.
Design of the Burroughs B1700.
In *Proc. of Fall Joint Computer Conf.*, pages 489-497. AFIPS, December, 1972.
In the B1700 each language has its own specialized microcode. Hardware provides for switching between microcode interpreters upon process switch. The Master Control Program is written in System Development Language, which has its own specialized microcode, and has a microprogrammed kernel for performing interrupt handling, scheduling, I/O processing, and virtual memory management.

[Wittie.L 80] Wittie, L.D. and Van Tilborg, A.M.
MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer.
IEEE Transactions on Computers C-29(12):1133-1144, December, 1980.

[Wulf.W 71] Wulf, W.A., Russell, D.B., and Habermann, A.N.
BLISS: A Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

[Wulf.W 74] Wulf, W.A., Cohen, E., Corwin, W., Jones, A.K., Levin, R., Pierson, C., and Pollack, F.
HYDRA: The Kernel of a Multiprocessor Operating System.
Communications of the ACM 17(6):337-345, June, 1974.

B.1.

Synchronizing Shared Abstract Types

(Revised Issue)

Peter M. Schwarz and Alfred Z. Spector

18 November 1983

Abstract

This paper discusses the synchronization issues that arise when transaction facilities are extended for use with shared abstract data types. A formalism for specifying the concurrency properties of such types is developed, based on dependency relations that are defined in terms of an abstract type's operations. The formalism requires that the specification of an abstract type state whether or not cycles involving these relations should be allowed to form. Directories and two types of queues are specified using the technique, and the degree to which concurrency is restricted by type-specific properties is exemplified. The paper also discusses how the specifications of types interact to determine the behavior of transactions. A locking technique is described that permits implementations to make use of type-specific information to approach the limits of concurrency.

Technical Report CMU-CS-83-163, Revision of CMU-CS-82-128

This research was sponsored by: the USAF Rome Air Development Center under contract F30602-81-C-0297; the US Naval Ocean Systems Center under contract number N66001-81-C-0484 N65; and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1 Introduction	1
2 Background	2
3 Dependencies: A Tool for Reasoning About Concurrent Transactions	4
3.1 Schedules	4
3.2 Dependencies and Consistency	5
3.3 Dependencies and Cascading Aborts	8
4 Specification of Shared Abstract Types	8
4.1 Directories	10
4.2 FIFO Queues	13
4.3 Queues Allowing Greater Concurrency	15
4.4 Proving the Correctness of Type Implementations	17
5 Orderability of Groups of Transactions	18
5.1 How the Specifications of Multiple Types Interact	19
5.2 Correctness of Transactions	20
6 A Technique for Synchronizing Shared Abstract Types	21
6.1 Type-Specific Locking	21
6.2 Directories	23
6.3 Strictly FIFO Queues	25
6.4 WQueues	26
6.5 Summary	27
7 Summary	28
8 Acknowledgments	29

1 Introduction

Transactions facilities, as provided in many database systems, permit the definition of *transactions* containing operations that read and write the database and that interact with the external world. The transaction facility of the database system guarantees that each invocation of a transaction will execute at most once (i.e., either commit or abort) and will be isolated from the deleterious effects of all concurrently executing transactions. To make these guarantees, the transaction facility manages transaction synchronization, recovery, and, if necessary, inter-site coordination. Many papers have been written about transactions in the context of both distributed and non-distributed databases [Bernstein 81, Eswaran 76, Gray 80, Lampson 81, Lindsay 79].

There are a number of ways in which transaction facilities could be extended to simplify the construction of many types of reliable distributed programs. Extensions that allow a wider variety of operations to be included in a transaction would facilitate manipulation of shared objects other than a database. Extensions that permit transaction nesting would facilitate more flexible program organizations, as would extensions allowing some form of inter-transaction communication of uncommitted data. Although the synchronization, recovery, and inter-site coordination mechanisms needed to support database transaction facilities are reasonably well understood, these mechanisms require substantial modification to support such extensions. For example, they must be made compatible with the abstract data type model and with general implementation techniques such as dynamic storage allocation.

Lomet [Lomet 77] considered some of the problems encountered in developing general-purpose transaction facilities, but more recently, much of the research in this area has been done at MIT. Moss and Reed have discussed nested transactions and other related systems issues [Moss 81, Reed 78]. As part of the Argus project, extensions to CLU have been proposed that incorporate primitives for supporting transactions [Liskov 82a, Liskov 82b]. Additionally, Weihl has considered transactions that contain calls on shared abstract types such as sets and message queues, and has discussed their implementation [Weihl 83a, Weihl 83b]. Transactions will also be available in the Clouds distributed operating system [Allchin 83].

This paper focuses on one important issue that arises when extending transaction facilities: the synchronization of operations on shared abstract data types such as directories, stacks, and queues. After a presentation of background material in the following section, Section 3 introduces some tools and notation for specifying shared abstract types. Section 4 describes three particular data types and uses the tools to specify how operations on these types can interact under conditions of concurrent access by multiple transactions. The specifications that are developed make explicit use of type-specific properties, and it is shown how this approach permits greater concurrency than standard techniques that do not use such information. Section

5 discusses how the specifications of individual types interact to determine global properties of groups of transactions. Section 6 proposes an extensible approach to locking that can be used for synchronization in implementations intended to meet these specifications. Finally, Section 7 summarizes the major points of this paper and concludes with a brief discussion of other considerations in the implementation of user-defined, shared abstract data types.

2 Background

Transactions aid in maintaining arbitrary application-dependent *consistency constraints* on stored data. The constraints must be maintained despite failures and without unnecessarily restricting the concurrent processing of application requests.

In the database literature, transactions are defined as arbitrary collections of database operations bracketed by two markers: *BeginTransaction* and *EndTransaction*. A transaction that completes successfully *commits*; an incomplete transaction can terminate unsuccessfully at any time by *aborting*. Transactions have the following special properties:

1. Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity*.

2. If a transaction completes successfully, the effects of its operations will never subsequently be lost. This property is usually called *permanence*.

3. If a transaction aborts, no other transactions will be forced to abort as a consequence. *Cascading aborts* are not permitted.

4. If several transactions execute concurrently, they affect the database as if they were executed serially in some order. This property is usually called *serializability*.

Transactions lessen the burden on application programmers by simplifying the treatment of failures and concurrency. Failure atomicity makes certain that when a transaction is interrupted by a failure, its partial results are undone. Programmers are therefore free to violate consistency constraints temporarily during the execution of a transaction. Serializability ensures that other concurrently executing transactions cannot observe these inconsistencies. Permanence and prevention of cascading aborts limit the amount of effort required to recover from a failure. Transaction models that do not prohibit cascading aborts are possible, but we do not consider them.

Our model for using transactions in distributed systems differs from this traditional model in several ways. The most important difference is that we incorporate the concept of an *abstract data type*. That is, information is stored in typed *objects* and manipulated only by *operations* that are specific to a particular

object type. The users of a type are given a *specification* that describes the effect of each operation on the stored data, and new abstract types can be implemented using existing ones. The details of how objects are represented and how the operations are carried out are known only to a type's implementor. Abstract data types grew out of the class construct in Simula [Dahl 72], and are supported in many other programming languages including CLU [Liskov 77], Alphard [Wulf 76], and Ada [Dept. of Defense 82], as well as in operating systems, e.g. Hydra [Wulf 74]. In our system model, transactions are composed of operations on objects that are instances of abstract types. Of particular interest are those objects that are not local to a single transaction. These are instances of *shared abstract types*.

We assume that the facilities for implementing shared abstract types and for coordinating the execution of transactions that operate on them are provided by a basic system layer that executes at each node of the system. This *transaction kernel* exports primitives for synchronization, recovery, deadlock management, and inter-site communication. In some ways, a transaction kernel is similar to the RSS of System R [Gray 81]. A transaction kernel, however, is intended to run on a bare machine and must supply primitives useful for implementing arbitrary data types, whereas the RSS has the assistance of an underlying operating system and only provides specialized primitives tailored for manipulating a database.

Another difference between our system model and the traditional transaction model is that we do not necessarily require that transactions appear to execute serially. Serializability ensures that if transactions work correctly in the absence of concurrency, any interleaving of their operations that is allowed by the system will not affect their correctness. But sometimes, serializability is too strong a property, and requiring it restricts concurrency unnecessarily. For example, it is usually unnecessary for two letters mailed together and addressed identically to appear in their recipient's mailbox together. However, serializability is violated if the letters do not arrive contiguously, because there is no longer the appearance that the sender has executed without interference from other senders. Thus, it may be desirable for some shared abstract types to allow limited non-serializable execution of transactions. This idea has also been investigated by Garcia-Molina [Garcia-Molina 83] and Sha et al. [Sha 83].

Serializability guarantees that an ordering can be defined on a group of transactions. If the transactions share some common objects, serializability requires that these objects be visited in the same order by all the transactions in the group. In the next section, a more general ordering property of transactions is defined, of which serializability is a special case. We will show that it is possible to prove that transactions work correctly in the presence of concurrency, even if they do not appear to execute serially.

In order to maintain the special properties of transactions in our model, the operations on shared abstract types that compose them must meet certain requirements. To guarantee the failure atomicity of transactions,

it must be possible to undo any operation upon transaction abort. Therefore, an *undo operation* must be provided for each operation on a shared abstract type. Recovery is not the main concern of this paper, and we will be considering undo operations only as they pertain to synchronization issues. Further discussion of recovery issues can be found in a related paper [Schwarz 83].

Operations on shared abstract types must also meet three synchronization requirements:

1. Operations must be protected from anomalies that could be caused by other concurrently executing operations on the same object. Freedom from these concurrency anomalies ensures that an invocation of an operation on a shared object is not affected by other concurrent operation invocations. This is the same property that monitors provide [Hoare 74].

2. To preclude the possibility of cascading aborts, operations on shared objects must not be able to observe information that might change if an uncommitted transaction were to abort. This may necessitate delaying the execution of operations on behalf of some transactions until other transactions complete, either successfully or unsuccessfully.

3. When a group of transactions invokes operations on shared objects, the operations may only be interleaved in ways that preserve serializability or some weaker ordering property of the group of transactions. The synchronization needed to control interleaving cannot be localized to individual shared objects, but rather requires cooperation among all the objects shared by the transactions.

Traditional methods for synchronizing access to an instance of a shared abstract type are designed solely to ensure the first goal: correctness of individual operations on an object. This paper is concerned with the second and third goals. We examine the problem of specifying the synchronization needed to achieve them, as well as the support facilities that the transaction kernel must provide to implementors of shared abstract types.

3 Dependencies: A Tool for Reasoning About Concurrent Transactions

This section introduces a theory that can be used to reason about the behavior of concurrent transactions. It allows the standard definition of serializability to be recast in terms of shared abstract types, and provides a convenient way of expressing other ordering properties. The theory is also useful in understanding cascading aborts.

3.1 Schedules

Schedules [Eswaran 76, Gray 75] can be used to model the behavior of a group of concurrent transactions. Informally, a schedule is a sequence of <transaction, operation> pairs that represents the order in which the component operations of concurrent transactions are interleaved. Schedules are also known as *histories* [Papadimitriou 77] and *logs* [Bernstein 79]. In some of the traditional database literature, the operations in schedules are assumed to be arbitrary; no semantic knowledge about them is available [Eswaran 76]. In this case, a schedule is merely an ordered list of transactions and the objects they touch:

T_1: O_1
T_2: O_1
T_2: O_2
T_1: O_2

In other work, operations are characterized as Read(R) or Write(W) [Gray 75], in which case the schedule includes that semantic information:

T_1: R(O_1)
T_2: R(O_1)
T_2: W(O_2)
T_1: R(O_2)

To analyze transactions that contain operations on specific shared abstract types, we will consider schedules in which these operations are characterized explicitly. For example, a schedule may contain operations to enter an element on a queue or to insert an entry into a directory. We call these *abstract schedules*, because they describe the order in which operations affect objects, regardless of any reordering that might be done by their implementation.[1] Given the initial state of a set of objects, an abstract schedule of operations on these objects, and specifications for the operations in the schedule, the result of each operation and the final state of the objects can be deduced. For instance, consider the following abstract schedule, which is composed of operations on Q, a shared object of type FIFO Queue. The operations QEnter and QRemove respectively append an element to the tail of a FIFO Queue and remove one from it's head. Assume Q to be empty initially.

T_1: QEnter(Q, X)
T_2: QEnter(Q, Y)
T_3: QRemove(Q)

From this abstract schedule and the initial contents of the Queue, one can deduce the state of Q at any point in the schedule. Thus one may conclude that the QRemove operation returns X, and that only Y remains on the Queue at the end of the schedule.

3.2 Dependencies and Consistency

By examining an abstract schedule, it is possible to determine what *dependencies* exist among the transactions in the schedule. The notation D: T_i:X $\rightarrow_O$ T_j:Y will be used to represent the dependency D formed when transaction T_i performs operation X and transaction T_j subsequently performs operation Y on some common object O. The object, transaction, or dependency identifiers may be omitted when they are unimportant. The set of ordered pairs $\{(T_i, T_j)\}$ for which there exist X, Y and O such that D: T_i:X $\rightarrow_O$ T_j:Y forms a relation, denoted $<_D$. If $T_i <_D T_j$, T_i *precedes* T_j and T_j *depends* on T_i, under the dependency D.

[1] In Section 4.4 we will define a second kind of schedule, the *invocation schedule*, which reflects the concurrency of specific implementations.

Examples of dependencies and their corresponding relations can be drawn from traditional database systems. For instance, consider a system in which no semantic knowledge, either about entire transactions or about their component operations, is available to the concurrency control mechanism. The only requirement is that each individual transaction be correct in itself: it must transform a consistent initial state of the database to a consistent final state. Under these conditions, only serializable abstract schedules can be guaranteed to preserve the correctness of individual transactions.

Since all operations are indistinguishable, only one possible dependency D can be defined: $T_1 <_D T_2$ if T_1 performs any operation on an object later operated on by T_2. Now, consider $<^*_D$, the transitive closure of $<_D$. A schedule is *orderable* with respect to $\{<_D\}$ iff $<^*_D$ is a partial order. In other words, there are no cycles of the form $T_1 <_D T_2 <_D \ldots <_D T_n <_D T_1$. In general, a schedule is orderable with respect to S, where S is a set of dependency relations, iff each of the relations in S have a transitive closure that is a partial order. The relations in S are referred to as *proscribed* relations, and we will use orderability with respect to a set of proscribed dependency relations to describe ordering properties of groups of transactions. Abstract schedules that are orderable with respect to a specified set of proscribed relations will be called *consistent* abstract schedules.

It can be shown that orderability with respect to $\{<_D\}$ is equivalent to serializability [Eswaran 76]. Given a schedule orderable with respect to $\{<_D\}$, a transaction T, and the set O of objects to which T refers, every other transaction that refers to an object in O can unambiguously be said either to precede T or to follow T. Thus T depends on a well-defined set of transactions that precede it, and a well-defined set of transactions depend on T. Each transaction sees the consistent database state left by those transactions that precede it, and (by assumption) leaves a consistent state for those that follow. The set of schedules for which $<^*_D$ is a partial order constitutes the set of consistent abstract schedules for a system that employs no semantic knowledge.

The scheme described above prevents cycles in the most general possible dependency relation, hence it maximally restricts concurrency. By considering the semantics of operations on objects, it is possible to identify some dependency relations for which cycles may be allowed to form. For example, consider a database with a Read/Write concurrency control. Such systems recognize two types of operations on objects: Read(R) and Write(W). Thus there are 4 possible dependencies between a pair of transactions that access a common object:

- D_1: T_i:R $\rightarrow_O$ T_j:R. T_i reads an object subsequently read by T_j.
- D_2: T_i:R $\rightarrow_O$ T_j:W. T_i reads an object subsequently modified by T_j.
- D_3: T_i:W $\rightarrow_O$ T_j:R. T_i modifies an object subsequently read by T_j.

- D_4: T_i:W $\rightarrow_O$ T_j:W. T_i modifies an object subsequently modified by T_j.

The earlier scheme, by not distinguishing between these dependencies, prevents cycles from forming in the dependency relation $<_D$, which is the union of all four individual relations. By contrast, Read/Write concurrency controls take into account the fact that R → R dependencies cannot influence system behavior. That is, given a pair of transactions, T_1 and T_2, and an abstract schedule in which both T_1 and T_2 perform a Read on a shared object, the semantics of Read operations ensure that neither T_1, T_2 nor any other transaction in the schedule can determine whether $T_1 <_{D_1} T_2$ or $T_2 <_{D_1} T_1$. Since these dependencies cannot be observed, they cannot compromise serializability, nor can they affect the outcome of transactions. We call dependencies meeting this criterion *insignificant*. Korth has also noted that when operations are commutative, their ordering does not affect serializability [Korth 83].

For the Read/Write case, the necessary condition for serializability can be restated as follows in terms of dependency relations: a schedule is serializable if it is orderable with respect to $\{<_{D_2 \cup D_3 \cup D_4}\}$ [Gray 75]. By allowing multiple readers, Read/Write schemes permit the formation of cycles in the $<_{D_1}$ dependency relation, and in relations that include $<_{D_1}$, while preventing cycles in the relation that is the union of $<_{D_2}$, $<_{D_3}$ and $<_{D_4}$. For example, consider the following schedules, which have identical effects on the system state:

T_1: R(O_1)	T_2: R(O_1)
T_2: R(O_1)	T_1: R(O_1)
T_1: W(O_1)	T_1: W(O_1)

In the first schedule, $T_1 <_{D_1} T_2$ and $T_2 <_{D_2} T_1$. Hence, there is a cycle in the relation $<_{D_1 \cup D_2}$, although $<_{D_2 \cup D_3 \cup D_4}$ is cycle-free. In the second schedule, the first two steps are reversed and neither cycle is present.

On the other hand, the following two schedules are not necessarily identical in effect:

T_1: R(O_1)	T_2: W(O_1)
T_2: W(O_1)	T_1: R(O_1)
T_1: W(O_1)	T_1: W(O_1)

In this case, the first schedule is not serializable because $T_1 <_{D_2} T_2$ and $T_2 <_{D_4} T_1$, thus forming a cycle in the relation $<_{D_2 \cup D_4}$, which is a sub-relation of $<_{D_2 \cup D_3 \cup D_4}$. T_1 observes O_1 before it is written by T_2, but the final state of O_1 reflects the Write of T_1 rather than T_2, implying that T_1 ran after T_2. The second schedule has no cycle and is serializable.

In summary, orderability with respect to a set of proscribed dependency relations provides a precise way to characterize consistent schedules. For a concurrency control that enforces serializability with no semantic knowledge at all about operations, the set of proscribed relations must contain $<_D$, which is equivalent to the union of every possible dependency relation. For a Read/Write database scheme, the set contains the $<_{R \rightarrow W \cup W \rightarrow R \cup W \rightarrow W}$ relation. When type-specific semantics are considered, type-specific dependency

relations can be defined for each type. In Section 4, dependencies are used to define *interleaving specifications* for various abstract types. These specifications provide the information needed to determine how an individual type can contribute toward maintaining a global ordering property such as serializability. If a specification guarantees orderability with respect to the union of all significant dependency relations for a given type, then it is strong enough to permit serializability. In general, however, more concurrency can be obtained when only weaker ordering properties are guaranteed. The way in which the interleaving specifications of multiple types interact to preserve global ordering properties is discussed in Section 5.

3.3 Dependencies and Cascading Aborts

Dependencies are also useful in understanding cascading aborts. A cascading abort is possible when a dependency forms between two transactions, the first of which is uncommitted. An abort by this uncommitted transaction may cascade to those that depend on it. Whether or not a cascade actually must occur depends on the exact type of dependency involved, and the properties of the object being acted upon. For example, consider the four general dependency relations that arise in Read/Write database systems. R → R dependencies are insignificant, and can never cause cascading aborts. This is analogous to the role of these dependencies in determining orderability. Likewise, R → W and W → W dependencies need not cause cascading aborts, because in both cases the outcome of the second transaction does not depend on data modified by the first[2]. By contrast, W → R dependencies represent a transfer of information between the two transactions. In the absence of any additional semantic information, it must be assumed that an abort of the first transaction will affect the outcome of the second, which must therefore also be aborted.

Once the dependencies that could lead to cascading aborts have been identified, their formation must be controlled. Stated in terms of abstract schedules: starting from the first of the two operations that form the dependency there must be no overlapping of the two transactions in the schedule, with the prior transaction in the dependency relation completing first. Such schedules will be called *cascade-free*. Note that some consistent schedules may not be cascade-free, and vice-versa.

4 Specification of Shared Abstract Types

This section focuses on the typed operations that make up transactions and discusses how to specify their local synchronization properties. The traditional specification of an abstract type describes the behavior of the type's operations in terms of preconditions, postconditions, and an invariant. This specification must be augmented in several ways to complete the description of a shared abstract type in our model. In the first place, the undo operation corresponding to each regular operation must be specified in terms of

[2] It may be necessary to control the formation of these dependencies anyway, if an insufficiently flexible recovery strategy is used.

preconditions, postconditions and the invariant. Specification of the undo operations themselves is not considered further in this paper. It is important to note, however, that the set of consistent abstract schedules defined by the interleaving specification for a type also implicitly includes schedules in which undo operations are inserted at all possible points after an operation has been performed but prior to the end of the invoking transaction. This reflects the assumption that it must be possible to undo any operation prior to transaction commitment. As will be shown in Section 4.3, this is especially important for types that do not attempt to enforce serializability of transactions.

The specification of a shared abstract type must also include a description of how operations on behalf of multiple transactions can be interleaved. This *interleaving specification* can be used by application programmers to describe their needs to prospective type implementors or to evaluate the suitability of existing types for their applications. The specification of a shared abstract type must also list those dependencies that will be controlled to prevent cascading aborts. This part of the specification is used mainly by the type's implementor.

When specifying how operations on a shared object may interact, the amount of concurrency that can be permitted depends in part on how much detailed knowledge is available concerning the semantics of the operations [Kung 79]. We have shown how concurrency controls that distinguish those operations that only observe the state of an object ("Reads") from those that modify it ("Writes") can achieve greater concurrency than protocols not making this distinction. To increase concurrency further while still providing serializability, one can take advantage of more semantic knowledge about the operations being performed [Korth 83]. Section 4.1 illustrates how this is done in specifying Directories, using the concepts and notation of the last section.

When enough concurrency cannot be obtained even after fully exploiting the semantics of the operations on a type, it is necessary to dispense with serializability and substitute orderability with respect to some weaker set of proscribed dependency relations. Sections 4.2 and 4.3 illustrate this by comparing a serializable Queue type with a variation that preserves a weaker ordering property.

Finally, Section 4.4 discusses how implementations may reorder operations to obtain even more concurrency, and the steps that type implementors must take to demonstrate the correctness of an implementation.

4.1 Directories

As a first example, consider a Directory data type that is intended to provide a mapping between text strings and capabilities for arbitrary objects. The usual operations are provided:

- DirInsert(dir, str, capa): inserts capa into Directory dir with key string str. Returns ok or duplicate key. The undo operation for DirInsert removes the inserted entry, if the insertion was successful.
- DirDelete(dir,str): deletes the capability stored with key string str from dir. Returns ok or not found. The undo operation for DirDelete restores the deleted capability, if the deletion was successful.
- DirLookup(dir, str): searches for a capability in dir with key string str. Returns the capability capa or not found. The undo operation is null, because DirLookup does not modify the Directory.
- DirDump(dir): returns a vector of <str,capa> pairs with the complete contents of the Directory dir. The undo operation for DirDump is null.

Suppose one wishes to specify the Directory type so as to permit serialization of transactions that include operations on Directories. One approach would be to model each DirInsert or DirDelete operation as a Read operation followed by a Write operation, and to model each DirLookup or DirDump operation as a Read operation. The Directory type could then be specified using the Read/Write dependency relations discussed previously.

The difficulty with using such limited semantic information is that concurrency is restricted unnecessarily. For example, suppose Directories have been implemented using a standard two-phase Read/Write locking mechanism. Consider the operation DirLookup(dir, "Foo"), which will be blocked trying to obtain a Read lock if another transaction has performed DirDelete(dir, "Fum") and holds a Write lock on the Directory object. The outcome of DirLookup(dir, "Foo") does not depend in any way on the eventual outcome of DirDelete(dir, "Fum") (which may later be aborted), or vice-versa, so this blocking is unnecessary. Because DirDelete(dir, "Fum") may be part of an arbitrarily long transaction, the Write lock may be held for a long time and severely degrade performance.

The unnecessary loss of concurrency in this example is not the fault of this particular implementation. It is caused by the lack of semantic information in the Directory specification. By using more knowledge about the operations, this problem can be alleviated. Instead of expressing the interleaving specification for this type in terms of Read and Write operations, the type-specific Directory operations can be employed to define dependencies and the interleaving specifications can be expressed in terms of these type-specific dependencies.

To keep the number of dependencies to a minimum, the operations for the Directory data type will be divided into three groups:

- Those that modify a particular entry in the Directory. DirInsert and DirDelete operations that succeed are in this class. These are Modify (M) operations.

- Those that observe the presence, absence, or contents of a particular entry in the Directory. DirLookup is in this class, as are DirInsert and DirDelete operations that fail. These are Lookup (L) operations.

- Those that observe properties of the Directory that cannot be isolated to an individual entry. DirDump is the only operation in this class that we have defined; an operation that returned the number of entries in the Directory would also be in this class. These are Dump (D) operations.

Note that in some cases operations that fail are distinguished from those that succeed. In addition to the operations and their outcomes, the dependencies also take into account data supplied to the operations as arguments or otherwise specific to the particular object acted upon. In the following list of dependencies, the symbols σ and σ' represent distinct key string arguments to Directory operations.

The complete set of dependencies for this type is:

- D_1: T_i:M(σ) → T_j:M(σ'). T_i modifies an entry with key string σ, and T_j subsequently modifies an entry with a different key string, σ'.

- D_2: T_i:M(σ) → T_j:M(σ). T_i modifies an entry with key string σ, and T_j subsequently modifies the same entry.

- D_3: T_i:M(σ) → T_j:L(σ'). T_i modifies an entry with key string σ, and T_j subsequently observes an entry with a different key string, σ'.

- D_4: T_i:M(σ) → T_j:L(σ). T_i modifies an entry with key string σ, and T_j subsequently observes the same entry.

- D_5: T_i:L(σ) → T_j:L(σ'). T_i observes an entry with key string σ, and T_j subsequently observes an entry with a different key string σ'.

- D_6: T_i:L(σ) → T_j:L(σ). T_i observes an entry with key string σ, and T_j subsequently observes the same entry.

- D_7: T_i:L(σ) → T_j:M(σ'). T_i observes an entry with key string σ, and T_j subsequently modifies an entry with a different key string σ'.

- D_8: T_i:L(σ) → T_j:M(σ). T_i observes an entry with key string σ, and T_j subsequently modifies the same entry.

- D_9: T_i:D → T_j:M(σ). T_i dumps the entire contents of the Directory, and T_j subsequently modifies an entry with key string σ.

- D_{10}: T_i:D → T_j:L(σ). T_i dumps the entire contents of the Directory, and T_j subsequently observes an entry with key string σ.

- D_{11}: T_i:M(σ) → T_j:D. T_i modifies an entry with key string σ, and T_j subsequently dumps the entire contents of the Directory.

- D_{12}: T_i:I.(σ) → T_j:D. T_i observes an entry with key string σ, and T_j subsequently dumps the entire contents of the Directory.

- D_{13}: T_i:D → T_j:D. T_i dumps the entire contents of the Directory and T_j subsequently dumps the Directory as well.

This list is long, but it is actually quite simple to derive. There is a family of dependencies for each pair of operation classes. The key to defining the specific dependencies is the observation that when two operations refer to different strings, the relationship between the transactions that invoked them is not the same as when they refer to identical strings. Those families of dependencies for which both operation classes take a string argument therefore have two members, corresponding to these two cases. The families for which one of the operation classes is Dump have only a single member. In general, insight into the semantics of a type is needed to define the set of possible dependencies.

Like the R → R dependency, many of the Directory dependencies are insignificant and cannot affect the outcome of transactions. Hence, they may be excluded from the set of proscribed dependencies for this type. The dependencies that may be disregarded are:

- Those for which neither operation in the dependency modifies the Directory object: D_6, D_{10}, D_{12} and D_{13}. These are directly analogous to the R → R dependency.

- Those for which the two operations in the dependency refer to different key strings: D_1, D_3, D_5, and D_7.

In terms of the remaining dependencies, the interleaving specification for Directories states that an abstract schedule involving Directories is consistent if it is orderable with respect to $\{<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}\}$. The abstract Directory thus defined behaves like a collection of associatively-addressed elements, with serializability preservable independently for each element. Transactions containing operations that apply to the entire Directory, such as DirDump, may also be serialized, as may those that refer to multiple elements or elements that are not present.

Only two of the Directory dependencies have the potential to cause cascading aborts. These are D_4 and D_{11}. In both cases, the first operation in the dependency modifies an entry and the second operation observes that modification.

4.2 FIFO Queues

Similar specifications can be developed for other data types. The FIFO Queue provides an interesting example. We will only consider two operations:

- QEnter(queue, capa): Adds an entry containing the pointer capa to the end of queue. The undo operation for QEnter removes this entry.

- QRemove(queue): Removes the entry at the head of queue and returns the pointer capa contained therein. If queue is empty, the operation is blocked, and waits until queue becomes non-empty. The undo operation for QRemove restores the entry to the head of queue.

In order to permit serialization of transactions that contain operations on strict FIFO Queues, and to prevent cascading aborts, numerous properties must be guaranteed. For instance:

- If a transaction adds several entries to a Queue, these entries must appear together and in the same order at the head of the Queue.

- Any entries added to a Queue by a transaction may not be observed by another transaction unless the first transaction terminates successfully.

- If two transactions each make entries in two Queues, the relative ordering of the entries made by the two transactions must be the same in both Queues.

It is very easy to destroy these properties if unrestricted interleaving of operations is allowed. For instance, if QEnter operations from different transactions are interleaved, the entries made by each transaction will not appear in a block at the head of the Queue.

In defining the dependencies for the Queue type, it is necessary, as it was in the case of Directories, to distinguish individual elements in the Queue. It is assumed that each element is assigned a unique identifier[3] when it is entered on the Queue. The symbols σ and σ' are used to represent the distinct identifiers of different elements, and the QEnter and QRemove operations are abbreviated as E and R respectively. The complete set of dependencies for Queues is:

- D_1: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}E(\sigma')$. T_j enters an element σ' into the queue Q after T_i has previously entered an element σ.

- D_2: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}R(\sigma')$. T_j removes element σ' after T_i entered element σ.

- D_3: $T_i{:}E(\sigma) \rightarrow_Q T_j{:}R(\sigma)$. T_j removes the element σ that was entered by T_i.

- D_4: $T_i{:}R(\sigma) \rightarrow_Q T_j{:}E(\sigma')$. T_j enters element σ' after T_i removed element σ.

---

[3] The identifier need not be globally unique, just unique among those generated for the particular Queue object.

- D_5: $T_i:R(\sigma) \rightarrow_Q T_j:R(\sigma')$. T_j removes element σ' after T_i removed element σ.

In a Read/Write synchronization scheme, QEnter must be modeled as a Write operation, and QRemove must be modeled as a Read followed by a Write. Recall that such a scheme must prevent cycles in the $<_{R \rightarrow W \cup W \rightarrow R \cup W \rightarrow W}$ dependency relation. In this case, preventing cycles in this general dependency relation is unnecessarily restrictive. Consider dependency D_2, which is formed when a transaction removes a Queue element after another transaction has previously entered a different Queue element. Neither of the transactions performing the operations can detect their ordering, nor can a third transaction. The same applies to dependency D_4, which is the inverse of D_2. As was the case for Directories, concurrency can be increased by disregarding insignificant dependencies.

To provide a strictly FIFO Queue, one must guarantee that abstract schedules are orderable with respect to the compound $<_{D_1 \cup D_3 \cup D_5}$ relation, but cycles may be permitted to form in relations that include D_2 or D_4 as long as this property is not violated. For example, consider the following schedule, in which two transactions operate on a Queue that initially contains {A, B}:

```
T1: QEnter(Q,X)
T2: QRemove(Q) returns A
T1: QEnter(Q,Y)
```

At step 2 of this schedule a D_2 dependency is formed, hence $T_1 <_{D_2} T_2$. At step 3, however, a D_4 dependency is formed with $T_2 <_{D_4} T_1$. Clearly a cycle exists in the compound relation $<_{D_2 \cup D_4}$. It is easy to create other examples of consistent abstract schedules that demonstrate a cycle in the basic $<_{D_2}$ (or $<_{D_4}$) relation, or in a compound relation formed from D_2 (or D_4) together with D_1, D_3 and D_5.

The dependency relations can also be used to characterize schedules susceptible to cascading abort. Dependency relation $<_{D_1}$ is similar to the W → W dependency. Since entries made by an aborted transaction can be transparently removed from the Queue, there is no danger of cascading abort. Relations $<_{D_3}$ and $<_{D_5}$ are more similar to W → R dependencies. In a D_3 dependency, information is transferred between the transactions in the form of the queue element σ; this dependency clearly can cause cascading aborts. A D_5 dependency can also cause cascading aborts, because the removal of an element by the first transaction affects which element is received by the second transaction.

While this definition of consistency for Queues is an improvement over a Read/Write scheme, it is still very restrictive of concurrency. It allows at most two transactions, one performing QEnter operations and one performing QRemove operations, to access a Queue concurrently. Unlike the Directory, the Queue is intended to preserve a particular ordering of the elements contained in it. A system based on serializable transactions guarantees that transactions can be placed in some order; by enforcing a particular order, data types such as queues (and stacks) restrict concurrency.

4.3 Queues Allowing Greater Concurrency

The preceding examples show how the use of semantic knowledge about operations on a shared abstract type permits increased concurrency. Once such knowledge is incorporated, the limiting factor in permitting concurrency becomes knowledge about the consistency constraints that the operations in a transaction attempt to maintain [Kung 79]. This knowledge concerns the semantics of groups of operations rather than individual ones. For example, a consistency constraint might state that every Queue entry of type A is immediately followed by one of type B. The potential for such constraints was the cause of the concurrency limitations observed above.

If it is possible to restrict the consistency constraints that a programmer is free to require, types guaranteeing ordering properties weaker than serializability may be acceptable. This may permit further increases in concurrency. A variation of the queue type can be used to demonstrate this.

One of the most common uses for a queue is to provide a buffer between activities that produce and consume work. Frequently, the exact ordering of entries on the queue is not important. What is crucial is that entries put on the rear of the queue do not languish in the queue forever; they should reach the head of the queue "fairly" with respect to other entries made at about the same time. A data type having this non-starvation property can be defined: the *Weakly-FIFO Queue* (WQueue for short). A similar type, the *Semi-Queue*, has been defined by Weihl [Weihl 83b].

The operations on WQueues and their corresponding undo operations are similar to those for Queues, but the interleaving specification for WQueues allows more concurrency. The dependencies for the WQueue type are the same as for the strict Queue. However, where the strict Queue required that consistent abstract schedules be orderable with respect to $\{<_{D_1 \cup D_3 \cup D_5}\}$, the WQueue permits cycles to occur in all the dependency relations save one: $<_{D_3}$. By allowing cycles in $<_{D_1}$, the interleaving of entries by multiple transactions becomes possible. Similarly, removing D_5 from the set of proscribed dependency relations permits WQRemove operations to be interleaved.

To take full advantage of the greater concurrency allowed by this interleaving specification, the semantics of WQRemove differ slightly from those of QRemove. If the transaction that inserted the headmost entry in the queue has not committed, that entry cannot be removed without risking the possibility of a cascading abort. Instead, WQRemove scans the WQueue and removes the headmost entry for which the inserting transaction has committed. If no such element can be found, any elements inserted by the transaction doing the WQRemove become eligible for removal. If neither a committed entry nor one inserted by the same transaction is available, the operation is blocked until an inserting transaction commits.

Modifying the semantics of WQRemove in this way does not destroy the fairness properties of the WQueue. No entry will remain in the WQueue forever if:

1. The transaction that entered it commits in a finite amount of time.

2. Transactions that remove it terminate after a finite amount of time.

3. Only a finite number of transactions remove the entry and then abort.

The behavior of the WQueue is best illustrated by example. In what follows, a WQueue is represented by a sequence of letters, with the left end of the sequence being the head of the WQueue. Lower case italic letters (*a*) are used to denote entries for which the WQEnter operation has not committed (i.e. the transaction that performed WQEnter is incomplete). Upper case bold letters (**A**) are used to represent entries that have not been removed and for which the entering transaction has committed. Upper case italic letters are used for entries that have been removed by an uncommitted WQRemove. Superscripts on entries affected by uncommitted operations identify the transaction that performed the operation.

Assume that the WQueue is initially empty. If transactions T_1 and T_2 perform WQEnter(WQ, a) and WQEnter(WQ, b) respectively, the WQueue's state becomes:

$\{a^1, b^2\}$

Since cycles in $<_{D_1}$ are permitted, T_1 may also add another entry, yielding:

$\{a^1, b^2, c^1\}$

If T_1 and T_2 both commit, the state becomes:

{A, B, C}

Note that the serializability of T_1 and T_2 has <u>not</u> been preserved. Now suppose that T_3 performs WQRemove and another transaction, T_4, removes two more elements:

$\{A^3, B^4, C^4\}$

If T_3 now aborts and T_4 commits, the final state becomes:

{A}

In this case, A and C have effectively been reversed, even though they were inserted initially by the same transaction! This example illustrates an important difference between shared abstract types that attempt to preserve serializability and those that do not: when a type permits non-serial execution of transactions, invoking an operation and subsequently aborting it is not necessarily equivalent to not invoking the operation at all. While we do not explicitly consider the undo operations in defining dependencies or interleaving specifications, the underlying assumption that aborts can occur at any time prior to commit implies that undo operations can be inserted at any point in a schedule between the invocation of an operation and the time at which the invoking transaction commits.

Another example indicates what happens when an uncommitted entry reaches the head of the Queue. Suppose the initial state is:

$\{a^5, b^6\}$

If T_6 commits but T_5 remains incomplete, the state becomes:

$\{a^5, B\}$

If T_7 removes an element at this time, B will be returned, leaving:

$\{a^5\}$

after T_7 commits. On the other hand, if T_5 commits after T_6, but before the remove by T_7, A will be returned even though its insertion was committed after B's.

To summarize the comparison between the WQueue and the ordinary Queue, note that two properties of the regular Queue have been sacrificed. First, strict FIFO ordering of entries is not guaranteed, because aborting WQRemove operations can reorder them. Second, transactions that operate on WQueues are not necessarily serializable with respect to all transactions in the system. Some other crucial properties, however, are preserved. The WQueue will not starve any entry, and it enforces an ordering of those transactions that communicate through access to a common element of the queue. This is ensured by orderability with respect to $\{<_{D_3}\}$. These modifications greatly increase concurrency, while still providing a data type that is useful in many situations.

4.4 Proving the Correctness of Type Implementations

Whereas the user of a type may employ the specified properties of abstract schedules (along with the rest of the type's specification) to reason about the correctness of transactions, the implementor of a type must prove the correctness of an implementation given the order in which operations are actually invoked. Real implementations may reorder the operations on an object to improve concurrency without changing the type's interleaving specification. Consider an implementation of the Queue type in which elements to be entered by a transaction are first collected in a transaction-local cache and entered as a block at end-of-transaction. This implementation allows any number of transactions to invoke the QEnter operation simultaneously, provided care is taken to serialize correctly transactions involving multiple Queues. By actually performing the insertions as a block, this implementation effectively reorders the individual QEnter operations to preserve consistency. It is possible to reorder QEnter operations in this way because QEnter does not return any information to its caller. Formation of any dependencies that might result from its invocation can therefore be postponed. The ultimate ordering of operations in the abstract schedule is determined by the implementation once all the QEnter operations to be performed by a given transaction are known. Thus, this implementation has the benefit of more knowledge about transactions than has the standard implementation.

Invocation schedules list operations in the order in which they are actually invoked, rather than in order of their abstract effects[4]. For example, the following is a possible invocation schedule for a Queue implemented using the block-insertion technique described above:

```
T2: QEnter(Q,Y)
T1: QEnter(Q,X)
T3: QRemove(Q)
```

If T_1 commits before T_2, the implementation reorders the two QEnter operations, resulting in the abstract schedule:

```
T1: QEnter(Q, X)
T2: QEnter(Q, Y)
T3: QRemove(Q)
```

The mapping between invocation schedules and abstract schedules is many-one; each invocation schedule implements exactly one abstract schedule, but an abstract schedule may be implemented by multiple invocation schedules. The synchronization mechanism used by an implementation determines a set of invocation schedules, called *legal schedules*, that are permitted by the implementation. The implementor must show that all legal invocation schedules map to consistent abstract schedules. To prevent cascading aborts as well, implementors must use a synchronization strategy that restricts the set of legal invocation schedules to those that map to abstract schedules that are in the intersection of the consistent and cascade-free sets.

5 Orderability of Groups of Transactions

The preceding section described how the standard specification of an abstract type, which only seeks to characterize the type's invariants and the postconditions for its operations, can be augmented with an interleaving specification that describes the local synchronization properties of objects. In this section we broaden our focus from the properties of the typed objects that are manipulated by transactions to the properties of entire transactions. We first examine how to generalize the definition of consistent abstract schedules to schedules that include operations on more than one object type, and then consider how ordering properties of groups of transactions can be used to show their correctness.

[4] It is assumed that the actual concurrent execution of the transactions can be modeled by a linear ordering of their component operations. This requires that the primitive operations be (abstractly) atomic. In the multiprocessor case, all linearizations of operations that could occur simultaneously yield distinct invocation schedules.

5.1 How the Specifications of Multiple Types Interact

Guaranteeing orderability with respect to the proscribed relations of a collection of individual types is not sufficient to ensure global ordering properties of transactions, such as serializability. Consider the following schedule, which contains transactions that operate both on Queues and Directories. Each of these types preserves orderability with respect to the union of all significant dependencies for the individual type, in order that transactions involving the type may potentially be serialized. However, this property alone does not guarantee serializability of the transactions. For example, the following schedule is not serializable:

```
T1: QEnter(Q,X)
T2: QEnter(Q,Y)
T2: DirInsert(D, "A", Z)
T1: DirDelete(D, "A")
```

Let $<_{Dir}$ stand for the $<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}$ relation, defined earlier for type Directory. Let $<_Q$ stand for the $<_{D_1 \cup D_3 \cup D_5}$ relation, defined earlier for Queues. Although the schedule is orderable with respect to $\{<_{Dir}, <_Q\}$, it is not serializable. To achieve serializability, the Queue and Directory types must cooperate to prevent cycles in the relation $\{<_{Dir \cup Q}\}$. The schedule is not orderable with respect to this compound dependency.

This example indicates how to generalize the definition of consistency to apply to abstract schedules containing operations on multiple types. Assume the interleaving specification for type Y_1 guarantees orderability with respect to $\{<_{D_1}\}$, the interleaving specification for type Y_2 guarantees orderability with respect to $\{<_{D_2}\}$, etc. The set of consistent abstract schedules involving types $Y_1, Y_2, \ldots Y_n$ is defined as those abstract schedules that are orderable with respect to $\{<_{D_1 \cup D_2 \cup \ldots \cup D_n}\}$: the union of the proscribed dependency relations of the individual types. A set of types whose implementations satisfy this property is called a set of *cooperative* types.

The need for cooperation among types does not necessarily imply that whenever a system is extended by the definition of a new type, the synchronization requirements of all existing types must be rethought. When designing a system, however, the implementors of cooperative types must first agree on a synchronization mechanism that is sufficiently flexible and powerful to meet all of their requirements. A poor choice of mechanism for fundamental building-block types will have an adverse effect on the entire system. Section 6 describes a mechanism based on locking that permits highly concurrent implementations of a large variety of shared abstract types.

5.2 Correctness of Transactions

When all of the types involved in a group of transactions cooperate to preserve an ordering property equivalent to serializability, it is easy to show that the correctness of transactions is not affected by concurrency. Because transactions are completely isolated from one another, a transaction can be proven correct solely on the basis of its own code and the assumption that the system state is correct when the transaction is initiated.

It is much more difficult to prove the correctness of transactions when they include operations on types that permit non-serializable interaction among transactions. One must consider the possible effects of interleaving each transaction with any other transaction, subject to the constraints of whatever ordering property is guaranteed by the collection of types. Nevertheless, in many practical situations, this task should not be insurmountable. We give two examples of situations where it is possible to make useful inferences about the behavior of transactions even though they preserve an ordering property weaker than serializability.

Users often invoke the DirDump operation on a Directory when they are "just looking around." In such cases, users would like to see a snapshot of the Directory's contents at an instant when the status of each entry is well defined, but they don't care what happens to the Directory thereafter. If all Directory operations attempt to enforce serializability, using DirDump in this way could greatly restrict concurrency. This problem can be alleviated by modifying the specification of the Directory type to permit limited non-serializable behavior.

Suppose dependency relations containing D_9: $T_i{:}D \rightarrow T_j{:}M(\sigma)$ are removed from the set of proscribed relations for the modified Directory type. That is, the interleaving specification for Directories only requires orderability with respect to $\{<_{D_2 \cup D_4 \cup D_8 \cup D_{11}}\}$ instead of $\{<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}\}$. Although this modified Directory allows non-serializable behavior, one can still guarantee that certain consistency constraints are not violated. For example, if a transaction replaces a group of entries in a Directory, one can still prove that no other transaction doing DirLookup operations will observe an incompatible collection of entries.

The WQueue of section 4.3 provides another example of a useful type that permits non-serializable interaction of transactions. Although the ordering property for WQueues is weaker than the one for strict Queues, some interesting properties can still be deduced based only on orderability with respect to $\{<_{D_3}\}$. Consider two transactions, T_1 and T_2, and two WQueues, Q_1 and Q_2. Suppose T_1 is intended to move all elements from Q_1 to Q_2 and T_2 is intended to move all elements from Q_2 to Q_1. If these transactions are run concurrently, the elements should all wind up in one WQueue or the other. This can be guaranteed only if $<_{D_3}$ is proscribed; otherwise elements could be shuffled endlessly between Q_1 and Q_2 and the transactions might never terminate.

6 A Technique for Synchronizing Shared Abstract Types

We have developed a formalism for specifying the synchronization of operations on shared abstract types, and interleaving specifications for some example types have been given. This section outlines a synchronization mechanism that can be used in implementations of these types. While we do not describe a particular syntax or implementation for this mechanism, we show how it can be used to prevent cascading aborts and control the interleaving of operations. We show how it provides the cooperation among types that is needed to preserve serializability or a weaker ordering property of a group of transactions. Implementation sketches for the shared abstract types specified in Section 4 are given as examples of its use.

As indicated in Section 4.4, the implementor of a type must take the following steps to demonstrate the correctness of an implementation:

1. characterize the set of legal invocation schedules, that is, those invocation schedules allowed by the synchronization mechanism used in the implementation.

2. give a mapping from invocation schedules to abstract schedules, and prove that the implementation carries out this mapping.

3. prove that every legal invocation schedule yields a consistent abstract schedule under this mapping.

This three-part task is simplest for implementations that are idealized in that they do not reorder operations on objects. Under these conditions, invocation schedules and abstract schedules are equivalent, and the second step in this process can be eliminated. The examples in this section discuss such idealized implementations of types.

6.1 Type-Specific Locking

The proposed synchronization technique is based on *locking*, which is used in many database systems to synchronize access to database objects. There are many variations on locking, but the same basic principle underlies them all: before a transaction is permitted to manipulate an object, it must obtain a *lock* on the object that will restrict further access to the object by other transactions until the transaction holding the lock releases it.

Locking restricts the formation of dependencies between transactions by restricting the set of legal invocation schedules. Whenever one transaction is forced to wait for a lock held by another, the formation of a dependency between the two transactions is delayed until the first transaction releases the lock. Under the well-known *two-phase locking* protocol [Eswaran 76], no transaction releases a lock until it has already claimed all the locks it will ever claim. This has the effect of converting potential cycles in dependency relations into

deadlocks instead. These can be detected, and because no dependencies have yet been allowed to form, either transaction can be aborted without affecting the other.

Locking is a conservative policy, because it delays the formation of any dependency that is part of a proscribed relation, not just those that eventually lead to cycles. This is not as significant a disadvantage as it might appear, however, because formation of those dependencies that transfer information (see Section 3.3) must be delayed anyway to prevent cascading aborts. In fact, the even more restrictive strategy of holding certain locks until end-of-transaction must often be employed to ensure that schedules are cascade-free. Furthermore, it is the conservative nature of locking protocols that makes them a suitable mechanism for sets of cooperative types. By preventing the formation of any dependencies local to a single object, cycles in proscribed relations that involve multiple types are automatically avoided without explicit communication between type managers. This is an important advantage, because it allows type managers to be constructed independently, as long as they correctly prevent the local formation of dependencies.

The chief disadvantage of many locking mechanisms is that they sacrifice concurrency by making minimal use of semantic knowledge about the objects being manipulated. The simplest locking schemes use only one type of lock, and hence cannot distinguish between significant and insignificant dependencies. Read/Write locking schemes use some semantic information, but are not flexible enough to take advantage of the extra concurrency specifiable in terms of type-specific dependencies. It has been shown [Kung 79] that two-phase locking is optimal under such conditions of limited semantic knowledge, but much more concurrency can be obtained if more semantic information is used. The locking technique described here generalizes the ideas behind Read/Write locking. It permits the definition of type-specific locking rules that reflect the interleaving specifications of individual data types. More restrictive type-specific locking schemes have previously been investigated by Korth [Korth 83].

Two observations can be made concerning type-specific dependencies. First, they specify the way in which type-specific operations on behalf of different transactions may be interleaved. Analogously, the generalized locking scheme requires the definition of type-specific *lock classes*, which correspond roughly to the operations on the type. Second, in addition to the operations, the dependencies reflect data supplied to the operations as arguments or data that is otherwise specific to the particular object acted upon. Therefore, an instance of a lock in the generalized locking scheme consists of two parts: the type-specific lock class and some amount of instance-specific data. It is the inclusion of data in the lock instance that differentiates our technique from Korth's. We use the notation {LockClass(data)} to represent an instance of a lock.

Once the lock classes for a type have been defined, a Boolean function must be given that specifies whether a particular new lock request may be granted as a function of those locks already held on the object. In

accordance with the practice in database literature, this function will be represented by a *lock compatibility table*. Only those locks held by other transactions need be checked for compatibility; a new lock request is always compatible with other locks held by the same transaction.

To complete the description of a type's locking scheme, one must specify the protocol by which each of the type's operations acquires and releases locks. Although two-phase locking can be used with type-specific locks, the locking protocol may also be type-specific. A uniform two-phase protocol is simplest to understand, but the added flexibility of type-specific protocols can allow increased concurrency. The exact nature of a type-specific protocol depends not only on the semantics of the type, but also on the particular representation and implementation chosen.

6.2 Directories

A simple idealized implementation of the Directory type specified in Section 4.1 illustrates the basics of type-specific locking. In this example, it is assumed that the Directory operations have been implemented in a straightforward fashion with no attempt at internal concurrency. It is further assumed that the operations act under the protection of a monitor or other mutual exclusion mechanism during the actual manipulation of Directory objects. Locking is used exclusively to control the sequencing of Directory operations on behalf of multiple transactions. The locking and mutual exclusion mechanisms cannot be completely independent, however, because mutual exclusion must be released when waiting for a lock within the monitor. This is a standard technique in systems that use monitors for synchronization [Hoare 74].

Because the mapping from invocation schedules to abstract schedules is trivial for this implementation, the second step of the validation process is eliminated. The discussion of the locking scheme for Directories therefore focuses on the first and third steps: informal characterization of the set of legal schedules, and comparison of this set with the set of consistent schedules.

As was noted in Section 4.1, the operations for the Directory data type can be divided into three groups:

- Modify operations, that alter the particular Directory entry identified by the key string σ.
- Lookup operations, that observe the presence, absence, or contents of the particular Directory entry identified by the key string σ.
- Dump operations, that observe properties of the Directory that cannot be isolated to an individual entry.

Corresponding to these groups, three lock classes can be defined:

- **{DirModify(σ)}**: To indicate that an incomplete transaction has inserted or deleted an entry with key string σ.

- {DirLookup(σ)}: To indicate that an incomplete transaction has attempted to observe the entry with key string σ.

- {DirDump}: To indicate that an incomplete transaction has performed a DirDump of the entire directory.

The lock compatibility table for Directories can be found in Table 1. Since there are a potentially infinite number of strings, the symbols σ and σ' are used to represent two arbitrary non-identical strings.

		Lock Held		
		DirModify(σ)	DirLookup(σ)	DirDump
Lock Requested	DirModify(σ)	No	No	No
	DirModify(σ')	OK	OK	No
	DirLookup(σ)	No	OK	OK
	DirLookup(σ')	OK	OK	OK
	DirDump	No	OK	OK

Table 1: Lock Compatibility Table for Directories

Each entry in this table reflects the nature of one of the type-specific dependency relations for Directories. Compatible entries represent dependency relations in which cycles are allowed to occur: for example, the entry in row 2, column 2 is "OK" because cycles are permitted in the $<_{M(\sigma) \rightarrow M(\sigma')}$ dependency relation. Incompatible entries reflect proscribed relations, such as the entry in row 1, column 2, which is due to the proscribed $<_{M(\sigma) \rightarrow M(\sigma)}$ relation.

The protocol used by the Directory operations for acquiring and releasing locks is as follows:

- DirInsert or DirDelete operations that specify the key string σ obtain a {DirModify(σ)} lock on the Directory. If the operation succeeds, the lock is held until end-of-transaction. If the operation fails, the lock is converted to a {DirLookup(σ)} lock, which is held until end-of-transaction.

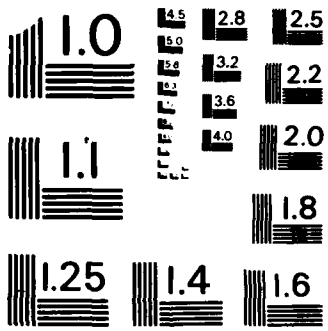
- DirLookup operations that specify the key string σ obtain a {DirLookup(σ)} lock on the Directory that is held until end-of-transaction.

- DirDump operations obtain a {DirDump} lock on the Directory that is held until end-of-transaction.

The following example demonstrates how the components of the locking scheme interact. Suppose a Directory D is initially empty. If a transaction T_1 performs the operation DirDelete(D, "Zebra"), this operation will fail by returning not found and leave a {DirLookup("Zebra")} lock on the Directory until the termination of T_1. Now suppose a second transaction, T_2, performs the operation DirInsert(D, "Zebra", capa). According to the protocol, DirInsert must first obtain a {DirModify("Zebra")} lock. Because the dependency relation $<_{L(\sigma) \rightarrow M(\sigma)}$ is proscribed, this lock is incompatible with the

AD-A169 754 DECENTRALIZED SYSTEM CONTROL(U) CARNEGIE-MELLON UNIV 4/4
PITTSBURGH PA DEPT OF COMPUTER SCIENCE
E D JENSEN ET AL. APR 86 CMU-CS-ARCHONS-83-1
UNCLASSIFIED RADC-TR-85-199 F30602-81-C-0297 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

{DirLookup("Zebra")} lock already held by T_1 (see row 1, column 3 of the compatibility table). Therefore, T_2 will be blocked. If T_1 subsequently becomes blocked while attempting to access an object already locked by T_2, a deadlock will occur. Both transactions are then blocked attempting to form dependencies that are part of proscribed relations. Although these relations may involve different objects, or even different types, a cycle in the union of the two relations is effectively prevented. This is exactly the behavior required to achieve consistency among cooperative types. On the other hand, if T_1 completes successfully the lock is released and the dependency of T_2 on T_1 is permitted to form. Since the $L(\sigma) \rightarrow M(\sigma)$ dependency cannot lead to cascading aborts, one may conclude (after the fact) that delaying T_2 was unnecessary.

By contrast, a transaction T_3 that performs the operation DirInsert(D, "Giraffe",capa) need not be blocked, because the $<_{L(\sigma) \rightarrow M(\sigma')}$ dependency relation is not proscribed. Accordingly, row 2, column 3 of the compatibility table indicates that a {DirModify("Giraffe")} lock is compatible with a {DirLookup("Zebra")} lock.

Although not a formal proof, this example characterizes the set of legal schedules permitted by the implementation, and shows how the lock classes, compatibility table, and locking protocol combine to guarantee that the legal schedules correspond to the consistent schedules defined in the last section. They capture the idea that, for this abstract data type, synchronization of access depends on the operations being performed, the particular entries in the Directory they attempt to reference, and their outcome. Because locks are on Directory objects, not components of directories, the technique also handles phantoms: entries that are mentioned in operations but are not present in the Directory.

6.3 Strictly FIFO Queues

Type-specific locking can also be used in implementations of the Queue data type of Section 4.2. As in the preceding example, assume a idealized implementation operating under conditions of mutual exclusion. To implement strictly FIFO Queues supporting only QEnter and QRemove operations, two lock classes are sufficient: {QEnter(σ)} and {QRemove(σ)}. As in the case of Directories, locks on Queues identify the particular entry to which the operation requesting the lock refers. Since Queue entries are not identified by key strings, it is assumed that at QEnter time, each element is assigned an identifier unique to the Queue instance. These identifiers correspond to those used in defining the dependency relations. Thus, a {QEnter(σ)} lock indicates that an element with identifier σ has been entered into the Queue by an incomplete transaction. Likewise, a {QRemove(σ)} lock indicates that the element with identifier σ has been removed form the Queue by an incomplete transaction.

The protocol for the Queue operations is:

- QEnter operations must obtain a {QEnter(σ)} lock, where σ is the newly-assigned identifier for the entry to be added. This lock is held until end-of-transaction.

- QRemove operations must obtain a {QRemove(σ)} lock, where σ is the identifier of the entry at the head of the Queue. This lock is held until end-of-transaction. Note that obtaining a {QRemove(σ)} lock does not necessarily imply that an entry σ is actually in the Queue, because the transaction that made the entry may have since aborted. If so, the QRemove operation must request a {QRemove(σ')} lock on the new headmost entry, σ'.

Table 2 shows the lock compatibility table for Queues. As usual, the symbols σ and σ' represent the identifiers of two different elements. Because the element identifiers are unique, certain situations (e.g. attempting to enter an element with the same identifier as an element already removed) cannot occur. The compatibility function is undefined in these cases, so the table entries are marked 'NA' for 'Not Applicable'.

		Lock Held	
		QEnter(σ)	QRemove(σ)
Lock Requested	QEnter(σ)	NA	NA
	QEnter(σ')	No	OK
	QRemove(σ)	No	NA
	QRemove(σ')	OK	No

Table 2: Lock Compatibility Table for Queues

The lock compatibility table reflects the limited concurrency of this type. Once a QRemove operation has retrieved the entry with identifier σ, some entry with identifier σ' becomes the head element of the Queue. But other transactions will be blocked trying to obtain the {QRemove(σ')} lock needed to remove it, until the first transaction completes. Multiple QEnter operations on behalf of different transactions interact in the same way. The incompatibility of {QRemove(σ)} with {QEnter(σ)} ensures that an uncommitted entry cannot be removed from the Queue, thereby eliminating a potential cause of cascading aborts.

6.4 WQueues

For a comparable idealized implementation of WQueues supporting only WQEnter and WQRemove, the same lock classes may be used as for FIFO Queues. The major difference between the two types shows up in the lock compatibility function, given by Table 3. To reflect the allowability of interleaved WQEnter operations by different transactions, the table entry in row 2, column 2 defines {WQEnter(σ)} and {WQEnter(σ')} locks to be compatible. Similarly, the entry in row 4, column 3 now permits multiple transactions to perform WQRemove operations. The only remaining restriction is the one in row 3, column 2 that prevents uncommitted entries from being removed. This prevents cycles in the proscribed $<_{E(\sigma) \rightarrow R(\sigma)}$ dependency relation and, because the lock is held until end-of-transaction, also prevents cascading aborts.

		Lock Held	
		WQEnter(σ)	WQRemove(σ)
Lock Requested	WQEnter(σ)	NA	NA
	WQEnter(σ')	OK	OK
	WQRemove(σ)	No	NA
	WQRemove(σ')	OK	OK

Table 3: Lock Compatibility Table for WQueues

The locking protocol for the WQueue operations is substantially the same as the one for the Queue operations. The only difference is that a WQRemove operation that is unable to obtain the required {WQRemove(σ)} lock on the element at the head of the WQueue does not block. Instead, WQRemove searches down the WQueue for some other element with identifier σ', for which a {WQRemove(σ')} lock can be obtained. This reflects the property of WQueues that permits elements farther down the WQueue to be removed when the head element is uncommitted. If no element can be found, the operation is blocked until an inserting transaction commits.

6.5 Summary

The examples in this section have shown how type-specific locking can be used for synchronization in implementations of several data types. The examples show how locking can be used to prevent cycles in proscribed dependency relations, including cycles containing several types of objects. They also indicate how locking can be used to prevent cascading aborts.

A full discussion of the syntax and implementation of type-specific locking mechanisms is beyond the scope of this paper. Further work is needed to determine the specific primitives required for definition of new object types, locking, unlocking, conditional locking, etc. Another area requiring further study is the relationship between the locking mechanism and other synchronization mechanisms that are used for mutual exclusion and to signal events. It appears, however, that implementation of a type-specific locking mechanism is often no more complex or expensive than implementations of standard locking. Unlike predicate locking schemes [Eswaran 76], the set of locks that apply to a particular object can easily be determined. It is also not difficult to determine what processes may be awakened in response to an event such as transaction completion.

7 Summary

This paper has been concerned with synchronizing transactions that access shared abstract types. In our model, four properties distinguish such types from others:

- Operations on them are permanent.
- They support failure atomicity of transactions.
- They do not permit cascading aborts.
- They contribute to preserving ordering properties of groups of transactions.

These properties are not independent, and the mechanisms that are used to achieve them are therefore related as well.

Schedules and dependencies are useful in understanding the interaction between concurrent transactions. The well-known consistency property of serializability can be redefined as a special case of orderability with respect to a dependency relation.. The specific dependency relation depends on how much semantic knowledge is available concerning operations on objects. When Read operations are distinguished from Write operations, serializability requires orderability with respect to a less restrictive dependency relation than when this distinction is not made. Dependencies can also be used to characterize schedules that are not prone to cascading aborts.

Additional type-specific semantic knowledge about operations can allow additional concurrency. The interleaving specifications for Directories and Queues developed in Sections 4.1 and 4.2 were stated in terms of orderability with respect to type-specific dependencies. To increase concurrency further, the WQueue sacrifices serializability while preserving orderability with respect to a less restrictive dependency. When several abstract types are combined in a transaction, orderability must be guaranteed with respect to the relation that is the union of the proscribed relations of the individual types.

Section 6 described a locking mechanism for implementing the synchronization required by the types described in Section 4. By allowing locks that consist of a type-specific lock class and instance-specific data, the mechanism provides a powerful framework for using type-specific semantics in synchronization. This mechanism is suitable for use in transactions containing multiple types, and it can also be used to prevent cascading aborts. The implementation of Directories shows how type-specific locking permits a uniform treatment of the problem of phantoms. Locks need not be directly associated with particular components of objects, which facilitates the separation of synchronization from other type representation issues. The examples of various Queue types show the mechanism's flexibility.

This paper has not provided a complete discussion of the issues involved in the specification and implementation of shared abstract types. For example, we have not discussed the construction of compound shared abstract types, which use other shared abstract types in their implementation. (However, Schwarz [Schwarz 82] gives an example of this.) In addition, we have hardly mentioned recovery considerations, though we believe logging mechanisms as described by Lindsay [Lindsay 79] can be extended to meet the needs of shared abstract types. Recovery is discussed more fully in a related paper [Schwarz 83]. Finally, we have not discussed specific algorithms for coping with deadlocks.

Clearly, the definition and implementation of shared abstract types is more difficult than the definition and implementation of regular abstract types. However, once these types are implemented, programmers can construct arbitrary transactions that invoke operations on the types. These transactions should greatly simplify the construction of reliable distributed systems. Though this paper has focused entirely on synchronization, we believe that this topic is central to understanding how transactions can be used as a basic building block in the implementation of distributed systems.

8 Acknowledgments

We gratefully acknowledge the helpful technical and editorial comments that were made by Dean Daniels, Cynthia Hibbard, Andy Hisgen, Bruce Lindsay, and Irving Traiger. We also gratefully acknowledge the Archons Project at Carnegie-Mellon University for providing the framework in which we conducted this research.

References

[Allchin 83] J. E. Allchin, M.S. McKendry.
Synchronization and Recovery of Actions.
In *Proc. of the Second Principles of Distributed Computing Conference*, pages 31-44. August, 1983.

[Bernstein 79] Philip A. Bernstein, David W. Shipman and Wing S. Wong.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering SE-5(3):203-216, May, 1979.

[Bernstein 81] Philip A. Bernestein and Nathan Goodman.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-221, June, 1981.

[Dahl 72] O.-J. Dahl and C. A. R. Hoare.
Hierarchical Program Structures.
In C. A. R. Hoare (editor), *A.P.I.C. Studies in Data Processing*. Volume 8: *Structured Programming*, chapter 3, pages 175-220. Academic Press, London and New York, 1972.

[Dept. of Defense 82]
Reference Manual for the Ada Programming Language
July 1982 edition, Dept. of Defense, Ada Joint Program Office, Washington, DC, 1982.

[Eswaran 76] K. P. Eswaran, James N. Gray, Raymond A. Lorie, I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Comm. of the ACM 19(11), November, 1976.

[Garcia-Molina 83]
Hector Garcia-Molina.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transactions on Database Systems 8(2):186-213, June, 1983.

[Gray 75] J. N. Gray, R.A. Lorie, G. R. Putzolu, and I. L. Traiger.
Granularity of Locks and Degrees of Consistency in a Shared Data Base.
IBM Research Report RJ1654, IBM Research Laboratory, San Jose, Ca., September, 1975.

[Gray 80] Jim Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, Ca., August, 1980.

[Gray 81] James N. Gray, et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys (2):223-242, June, 1981.

[Hoare 74] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Comm. of the ACM 17(10):549-557, October, 1974.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), Jaunary, 1983.

[Kung 79] H. T. Kung and C. H. Papadimitriou.
An Optimality Theory of Concurrency Control for Databases.
In *Proceedings of the 1979 SIGMOD Conference.* ACM, Boston, MA., May, 1979.

[Lampson 81] Butler W. Lampson.
Atomic Transactions.
In G. Goos and J. Hartmanis (editors). *Lecture Notes in Computer Science.* Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course,* chapter 11, pages 246-265. Springer-Verlag, 1981.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

[Liskov 77] B. Liskov, A. Snyder, R.Atkinson, C. Schaffert.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8), August, 1977.

[Liskov 82a] Barbara Liskov.
On Linguistic Support for Distributed Programs.
IEEE Trans. on Software Engineering SE-8(3):203-210, May, 1982.

[Liskov 82b] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages,* pages 7-19. Albuquerque, NM, January, 1982.

[Lomet 77] David B. Lomet.
Process Structuring, Synchronization, and Recovery Using Atomic Actions.
ACM SIGPLAN Notices 12(3), March, 1977.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.

[Papadimitriou 77]
C. H. Papadimitriou, P. A. Bernstein, and J. B. Rothnie.
Some Computational Problems Related to Database Concurrency Control.
In *Proceedings of the Conference on Theoretical Computer Science.* Waterloo, Ont., August, 1977.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.

[Schwarz 82] Peter M. Schwarz.
Building Systems Based on Atomic Transactions.
1982.
PhD. Thesis Proposal, Carnegie-Mellon University.

[Schwarz 83] Peter M. Schwarz, Alfred Z. Spector.
Recovery of Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-83-151, Carnegie-Mellon University, Pittsburgh, PA, October, 1983.

[Sha 83] Lui Sha, E. Douglas Jensen, Richard F. Rashid, J. Duane Northcutt.
Distributed Co-operating Processes and Transactions.
In *Proceedings ACM SIGCOMM Symposium.* 1983.

[Weihl 83a] William E. Weihl.
data Dependent Concurrency Control and Recovery.
In *Proc. of the Second Principles of Distributed Computing Conference*, pages 73-74. August 1983.

[Weihl 83b] W. Weihl, B. Liskov.
Specification and Implementation of Resilient, Atomic Data Types.
In *Symposium on Programming Language Issues in Software Systems.* June, 1983.

[Wulf 74] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack.
HYDRA: The Kernel of a Muiltiprocessor Operating System.
Communications of the ACM 17(6):337-345, June, 1974.

[Wulf 76] W. A. Wulf, R. L. London, M. Shaw.
An Introduction to the Construction and Verification of Alphard Programs.
IEEE Transactions on Software Engineering SE-2(4), December, 1976.

B.2.

Transactions: A Construct for Reliable Distributed Computing

Preliminary Draft

Alfred Z. Spector and Peter M. Schwarz

Abstract

Transactions have proven to be a useful tool for constructing reliable database systems and are likely to be useful in many types of distributed systems. To exploit transactions in a general purpose distributed system, each node can execute a transaction kernel that provides services necessary to support transactions at higher system levels. The transaction model that the kernel supports must permit arbitrary operations on the wide collection of data types used by programmers. New techniques must be developed for specifying the synchronization and recovery properties of abstract types that are used in transactions. Existing mechanisms for synchronization, recovery, deadlock management and communication are often inadequate to implement these types efficiently, and they must be adapted or replaced.

Technical Report CMU-CS-82-143

This work was sponsored in part by: the USAF Rome Air Development Center under contract F30602-81-C-0297; the US Naval Ocean Systems Center under contract number N66001-81-C-0484; the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539; and the IBM Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

1. Introduction

Distributed computing systems are potentially reliable, because the redundancy and autonomy present in them permit failures to be masked or localized. A major challenge in distributed computing research is to realize this potential without incurring intolerable penalties in complexity, cost, or performance. Consequently, there is currently great interest in general-purpose methodologies and practices *that simplify* the construction of efficient and robust distributed systems. This paper discusses a methodology based on *transactions* and includes a survey of considerations in the design of a *transaction kernel*: an abstract machine that supports transactions.

Transactions were originally developed for database management systems, to aid in maintaining arbitrary application-dependent *consistency constraints* on stored data. The constraints must be maintained despite failures and without unnecessarily restricting the concurrent processing of application requests.

In the database literature, transactions are defined as arbitrary collections of operations bracketed by two markers: *BeginTransaction* and *EndTransaction*, and have the following special properties:

- Either all or none of a transaction's operations are performed. This property is usually called *failure atomicity*.
- If a transaction completes successfully, the results of its operations will never subsequently be lost. This property is usually called *permanence*.
- If several transactions execute concurrently, they affect the database as if they were executed serially in some order. This property is usually called *serializability*.
- An incomplete transaction cannot reveal results to other transactions, in order to prevent *cascading aborts* if the incomplete transaction must subsequently be undone.

Transactions lessen the burden on application programmers by simplifying the treatment of failures and concurrency. Failure atomicity makes certain that when a transaction is interrupted by a failure, its partial results are undone. Programmers are therefore free to violate consistency constraints temporarily during the execution of a transaction. Serializability ensures that other concurrently executing transactions cannot observe these inconsistencies. Prevention of cascading aborts limits the amount of effort required to recover from a failure.

Database management systems are not the only ones that must assure the consistency of stored data despite failures and concurrency. Various ad hoc techniques have evolved for this purpose. For example, TOPS-10 [Digital Equipment Corporation 72] and numerous other file systems permit atomic updates to a single disk

file. This technique lacks flexibility and generality, however, and leads to unnecessary restrictions on concurrency.

Considerable research effort is currently being expended towards extending the utility of transactions beyond database applications. At MIT, the Argus project [Liskov 82a] is adding transaction facilities to the CLU language. Transactions will also be available in the Clouds distributed operating system [Allchin 82].

At Carnegie-Mellon, we are exploring the idea of implementing a *transaction kernel* on each node of a distributed system. A transaction kernel is a basic system component that supplies primitives for supporting transactions and the shared abstract data types on which they operate. Complex, costly, and redundant error recovery mechanisms could be avoided elsewhere, if this facility were available. A transaction kernel should also lead to compatible structuring of the various systems that use it, simplifying their interconnection.

This report is an overview of recent research on transaction systems, and surveys issues that arise in developing a transaction kernel. We consider the extension of transactions to general programming and discuss how a transaction kernel should facilitate data abstraction. Subsequent sections examine what we believe to be the central issues in building a transaction kernel: synchronizing access to shared abstract types without unnecessarily restricting concurrency, managing deadlocks, recovering from failures, and communicating efficiently between sites. For more on the extended use of transactions, we refer the reader to recent reports by Liskov, Allchin, Jacobson, and ourselves [Allchin 82, Liskov 82b, Liskov 82a, Jacobson 82, Schwarz 82].

2. Extensions to the Transaction Model

A construct that gives programmers a uniform strategy for treatment of failures, controls interaction between concurrently executing processes, and ensures permanence of operations should simplify the production of reliable distributed systems. Except for database applications, however, the utility of transactions has not been widely demonstrated. Lomet hypothesized that transactions would be useful for general programming [Lomet 77], but the literature includes sketches of only a few non-database systems based on transactions [Liskov 82a, Allchin 82, Gifford 79, Daniels 82].

The traditional transaction model, as described by Gray [Gray 80], was designed primarily for understanding database management applications. It must be extended to model the additional requirements imposed by general-purpose distributed systems. For instance, real-time systems may require real-time synchronization of the participants in transactions (see Section 6). File and mail systems that are both highly available and highly reliable are also difficult to implement unless constructs not in the traditional transaction model are used. Their transactions are more complex than those in database systems, and their performance

requirements are potentially higher. Gray also comments on the limitations of the traditional transaction model [Gray 81a].

Database systems, and their transaction mechanisms, do not fully support the abstract data types that are required in more general systems. In a database, the basic unit of information is the typed record, which can be aggregated into indexed files. The only operations on records are read and write, and only operations such as insert, lookup, or sort are defined for files.

Systems that encourage data abstraction must be more flexible. They must permit the definition of arbitrary object types with corresponding sets of type-specific operations. They must also allow new object types to be implemented by combining existing ones, and the resulting types should appear to their users as primitive types. Rather than a sequence of reads and writes on records, a transaction becomes a hierarchy of typed operations on objects. Transactions can be nested, if some of the operations in the hierarchy are themselves implemented with transactions.

Nested transactions are also useful for controlling the interaction of multiple processes within a single transaction, or salvaging partial results when a transaction aborts. For example, some real-time applications employ fairly lengthy transactions. If aborting such transactions and restarting them from the beginning would cause intolerable delays, the transactions must instead fall back to intermediate *save points* [Gray 81c]. See Reed's and Moss' theses for more about nested transactions [Reed 78, Moss 81].

In database systems, application programmers do not have to specify the consistency constraints that they wish transactions to preserve. By guaranteeing serializability of all transactions, database transaction mechanisms assure that any consistency constraint preserved when a transaction runs in isolation will also be preserved when transactions run concurrently. The transaction manager must delay or abort transactions as necessary to make this guarantee. If the system were aware of the specific consistency constraints that transactions were intended to maintain, it could use this extra information in deciding whether or not to delay or abort transactions. Avoiding unnecessary delays and aborts would improve performance. Semantic knowledge about individual types, their operations, and their implementations could also be used to make better-informed decisions regarding concurrent access to objects. A transaction mechanism efficient enough for use in general-purpose distributed systems must be flexible enough to allow such use of semantic information to achieve greater concurrency.

Our approach is to focus on the individual shared abstract types that programmers use in constructing transactions. In addition to the traditional properties of abstract types, these types can be characterized by their synchronization and recovery properties. The specification of these properties defines the types' exact

behavior under conditions of concurrency or failure. Assuming different types cooperate in a reasonable fashion, the specifications allow programmers to determine whether particular types will meet their needs. Sections 3 and 5 discuss in detail the synchronization and recovery properties of shared abstract types.

We have found types with specialized synchronization and recovery properties to be useful in designing a highly available message system. For example, message repositories, which are replicated on several sites, are highly specialized shared abstract objects with unique sets of operations. In addition to reading and writing messages, special operations permit out-of-date message repositories to "catch up" with current ones. The recovery and synchronization properties of these operations are type-specific and must be carefully specified and analyzed.

3. Synchronization

In a transaction-based system, synchronization is important to both the specification and the implementation of shared abstract types. Traditional methods for synchronizing access to objects (e.g., monitors [Hoare 74]) just prevent concurrent operations on a particular object from interfering with one another. Maintaining consistency constraints that encompass groups of objects necessitates additional synchronization. However, the mechanisms that enforce this additional synchronization must not unnecessarily restrict concurrency. Because transactions are arbitrarily large collections of operations, a synchronization action that is in force over the entire scope of a transaction can potentially degrade performance more severely than a synchronization action that only affects a single operation.

One approach to synchronization in a general transaction-based system is to classify each operation on an abstract type as either a Read or a Write. A two-phase Read/Write locking scheme [Eswaran 76] ensures serializability and, if locks are held until end-of-transaction, prevents cascading aborts as well. However, such techniques for managing concurrency make minimal use of semantic knowledge about the objects that transactions manipulate, and therefore they may prevent or delay operations unnecessarily.

For example, consider two transactions that each insert a new entry in a directory object. Since the insertion operations modify the directory object, one must classify them as Write operations. The standard rules for Read/Write locking prohibit modification of an object by more than one incomplete transaction. The system would therefore delay the second insertion until the transaction making the first one either committed or aborted. Closer examination of the semantics of insertion reveals that this is unnecessary if the two insertions specify different keys. A synchronization mechanism that could use this extra knowledge could achieve greater concurrency.

Similarly, specifying serializability as the goal of a transaction synchronization strategy reflects a limited use

of semantic knowledge. Serializability makes sure that any invariant preserved by an individual transaction will also be preserved when transactions execute concurrently. This guarantee is frequently too strong. For instance, consider a queue that buffers units of work between activities that produce and consume them. Serializing the transactions that operate on the buffer queue groups together all entries made by a single transaction, in order to enforce their consecutive removal. In many applications, ordering of entries in the buffer is not crucial as long as entries for which the inserting transaction has committed eventually reach the head and can be removed. Entries inserted by incomplete transactions must not be removed, however, so that cascading aborts cannot occur. As in the preceding example, using more semantic knowledge about the object and its intended purpose can lead to greater concurrency.

Many authors [Eswaran 76, Kung 79, Allchin 82, Garcia-Molina 82, Sha 83] have observed that using semantic knowledge can increase concurrency. While Garcia and Sha consider the properties of entire transactions, we are concentrating on the semantics of operations on individual types. To exploit this approach, one must first be able to specify precisely and concisely how a type behaves under conditions of concurrent access by multiple transactions. Prospective users need such a means of specification to define their own requirements and to compare them with the properties of available types. We have investigated *dependencies* as a tool for this purpose. Dependencies were originally used in database research for proving the correctness of two-phase locking protocols [Eswaran 76, Gray 75]. A dependency exists between any two transactions that perform an operation on a common object, and the dependency defines the order in which the two transactions operate on the object.

One can prove that if the transitive closure of all the dependencies among transactions forms a partial order, then the execution of the transactions is serializable [Eswaran 76]. If the transitive closure contains cycles, the ordering of transactions is ambiguous. Not all dependencies are equivalent, however. For example, the semantics of the Read operation tell us that the order in which two transactions read a common object has no effect on the transactions' outcome. Even though the transitive closure of all dependencies has cycles, disregarding these meaningless dependencies and recomputing the transitive closure may result in a partial order of the transactions. In general, a group of transactions is *orderable* with respect to a particular group of *proscribed* dependencies if the transitive closure of the proscribed dependencies yields a partial order. Serializability in a database with Read/Write locking can be defined in these terms as orderability with respect to all dependencies except those for which both operations are Reads [Gray 75].

In a general-purpose system with arbitrary shared abstract types, a set of proscribed dependencies must be defined for each type. Semantic knowledge about individual types can be used in constructing this set, to achieve high concurrency while still helping the programmer to preserve consistency. For instance, the proscribed set of dependencies for directories would not include dependencies between transactions operating

on entries with different keys. Like dependencies in which both operations are Reads, these dependencies cannot affect consistency. To specify a queue type for which grouping of elements by inserting transaction is not assured, dependencies between transactions performing the insert operation can be removed from the proscribed dependency set.

When a transaction accesses several objects of different types, the types must cooperate to maintain global consistency. In addition to guaranteeing orderability with respect to the proscribed dependency sets of the individual types, the transaction manager must also preserve orderability with respect to the union of the proscribed dependency sets.

Dependencies can also be used to specify which operations must be delayed to prevent potential cascading aborts. Whenever a dependency is about to form between two incomplete transactions, the second transaction may have to be delayed in case the first one aborts. The decision whether or not to delay depends on the exact dependency being formed. Analogous to the proscribed dependency set, each type must specify a deferred dependency set that determines the circumstances under which operations will be delayed until a prior transaction commits or aborts. Usually, dependencies that represent a transfer of information between the two transactions must be deferred. A more extensive treatment of the dependency technique, including detailed examples, can be found in a related paper [Schwarz 82].

Shared abstract types can be divided into three categories based on their synchronization behavior. The categories are listed in order of increasing potential for concurrent access, and each properly includes the preceding ones.

1. Types that serialize access to objects. These types can use semantic knowledge to permit greater concurrent access to an object without losing the advantages of serializability. The directory that allows concurrent operations on entries with different keys is in this category. The proscribed dependency sets for these types includes all dependencies that have a detectable effect on transaction outcomes.

2. Types that do not permit incomplete transactions to reveal their results to other transactions. Since transactions are not necessarily serializable, this strategy does not guarantee arbitrary consistency constraints, but can lead to higher concurrency while still preserving properties that are crucial to the purpose of the type. The queue that does not guarantee grouping by inserting transaction is in this category. Some dependencies that affect transaction outcomes can be excluded from these types' proscribed dependency sets.

3. Types with arbitrary synchronization policies. Incomplete transactions that operate on objects of these types may reveal data to other transactions; it is assumed that these data are acceptable (i.e., will not cause cascading aborts) even if the revealing transaction subsequently aborts. An update

that is used only as a "hint" can be revealed, for instance. Even if another transaction reads the hint while it has an incorrect value, no fatal error will occur. In terms of dependencies, the deferred dependency sets for these types may exclude some dependencies that transfer information.

Given dependencies as a means of specification, a second key to achieving efficient synchronization by utilizing semantic knowledge is the definition of a synchronization mechanism flexible enough to implement a wide variety of shared abstract types. We have examined using *type-specific locking* as this mechanism. Bernstein, Goodman, and Lai [Bernstein 81] discuss some of this method's basic principles. Korth [Korth 83] has described a type-specific approach to locking based on commutativity of operations, which employs a hierarchy of locks to allow variable-granularity locking. Weihl, in connection with the Argus Project, has described *crowds*: an alternative synchronization mechanism for exploiting type-specific semantics [Weihl 81]. Transactions must join a crowd before accessing an object and only leave the crowd when the transaction is complete. Type-specific rules determine whether a transaction should be admitted a crowd or be forced to wait until some other conflicting transaction leaves.

A set of basic principles underlies all locking schemes. Before a transaction manipulates an object, it must obtain a *lock* on the object. Possession of the lock restricts further access to the object by other transactions, until it is released. Locking mechanisms thus control the formation of dependencies among transactions. Whenever one transaction waits for a lock held by another, formation of a dependency between the two transactions is delayed until the lock is released. The protocol for acquiring and releasing locks ensures that if the dependency would become part of a cycle in the transitive closure of a set of proscribed dependencies, a deadlock results and the cycle never forms.

The simplest locking mechanisms have only one kind of lock, regardless of the type of the object to be locked or the operation to be performed. This form of locking uses no semantic knowledge, and cannot distinguish between proscribed and non-proscribed dependencies. Many database systems use a locking mechanism that provides two *lock classes*, Read and Write. Operations that modify an object must first obtain a Write lock, whereas operations that merely reference an object's value need only obtain a Read lock. The rules for obtaining locks specify that multiple transactions may simultaneously hold Read locks on an object, but holding a Write lock reserves the object exclusively for one transaction. By making this coarse distinction among different kinds of operations, Read/Write locking uses limited semantic information to permit some cyclic dependencies while prohibiting others. This yields greater concurrency without compromising consistency.

Type-specific locking generalizes the ideas behind Read/Write locking. Instead of dividing all operations

into two broad classes, the implementor of each type can define appropriate type-specific lock classes and associated rules for acquiring and releasing locks. The rules specify the kind(s) of lock required by each of the type's operations and which kinds of locks are compatible with each other. By tailoring the locking strategy to suit a specific type and implementation, type-specific locking preserves only what is promised by the type's specification. A large amount of semantic information about both the specification and the implementation of the type can be used in deciding whether an operation must be delayed or prevented.

Additional research is needed to determine specific primitives for locking, unlocking, definition of new object types, etc. For example, data stored in an object or supplied as an argument to an operation is sometimes crucial in determining the compatibility of two operations. Recall that insert operations on directories are compatible only if they refer to entries with different keys. Type-specific locking primitives must permit the association of auxiliary information with locks on objects.

A related paper by Schwarz and Spector [Schwarz 82] contains further details and examples of type-specific locking. It appears that implementations of type-specific locking mechanisms will be reasonably simple and, in order to understand their details more completely, we are building one using directories as a sample shared abstract type.

4. Deadlock

One must consider the possibility of deadlock in any system where processes may wait for dynamically allocated resources. In a transaction, the resources are the objects that the transaction accesses. There are many strategies for coping with deadlocks, but it is not clear which are most appropriate for transaction-based systems with arbitrary shared abstract types.

One approach is to impose a global ordering on all system resources, and force all transactions to obtain resources according to this ordering. This method is unsuitable, because it does not allow transactions that access a data-dependent collection of objects. When the system initiates a transaction, it must know a priori all the resources the transaction will need. Another technique uses timestamps on transactions or objects to avoid deadlock [Rosenkrantz 78, Reed 78]. A third approach to the problem is to allow deadlocks, subsequently detect them, and ultimately resolve them by selecting a transaction to abort. Either timeouts or an algorithm that analyzes waiting transactions can be used for detection. Unfortunately, employing timeouts causes the timing behavior of an abstract data type's implementation to become a critical aspect of the type's specification. In either case, detection and resolution of deadlocks could become a bottleneck that would constrain performance.

Arbitrary type-specific locking protocols can cause another problem. If a protocol allows the release of

some locks prior to end-of-transaction, it may be necessary to re-acquire them later to process an abort. Re-acquisition violates the common simplifying assumption that aborting a transaction never requires additional resources. Deadlocks can therefore occur during abort processing, and the standard approach of aborting a waiting transaction cannot resolve them.

There has been fairly little formal analysis of the relationships between the probability of waiting or deadlock and such factors as degree of multi-programming, number of operations in a transaction, and size of the shared database. However, Gray et al. and Lin et al. [Gray 81b, Lin 82] have each modeled both the probability of waiting for a lock request and the probability of deadlock in two phase locking protocols, and they conclude that both probabilities rise with the degree of multiprogramming. They also report that the probabilities of deadlock and waiting rise more than linearly in the number of operations per transaction. These pessimistic conclusions are based on very simple models. They must be adapted if they are to represent accurately the behavior of transactions that access a hierarchically structured graph of typed objects. It seems reasonable, however, to conclude that if many transactions frequently access small groups of objects, contention and deadlock would become serious problems.

To summarize, the problems of deadlock are exacerbated in general-purpose transaction-based systems. Further research is needed to examine the applicability of traditional solutions, and to determine the tradeoffs among those solutions in this environment. This research may yield variations on the traditional solutions, or demonstrate the need for new algorithms specifically designed for shared abstract types. For an example of a new approach to deadlock avoidance, see Korth's hierarchical variable-granularity locking protocol [Korth 81], which uses *edge locks*.

5. Recovery

Recovery is the process of restoring consistency after a failure. Recovery properties can be used like synchronization properties to classify types, and different recovery techniques are appropriate for different classes of types:

Some types have operations that are uninvertible. Gray has called such types *real* [Gray 80], because their operations correspond to events in the "real" world that are either unrepeatable or irreversible. An operation that causes a banking terminal to dispense cash is an example of an uninvertible update. These operations must be deferred until the invoking transaction commits.

Other types can be characterized by two properties of their operations: failure atomicity and permanence. Failure-atomic operations are always undone upon transaction abort, and if all operations in a transaction are failure-atomic then the entire transaction will be failure-atomic. Failure-atomic operations must be undone

both when transactions abort during normal processing and when transactions are interrupted by failures. After a failure, recovery must identify and then abort any transactions that were in progress. Operations that are not failure-atomic are useful for implementing hints efficiently. As discussed in Section 3, incorrect hints do not cause fatal errors or loss of consistency.

Permanent operations are never undone once a transaction has committed. Guaranteeing permanence, unlike failure atomicity, requires that the system store some information in a failure-resilient manner. This is potentially expensive, and there are many types that do not need to survive failures. The cost of reconstructing an object's state from other information after a failure can be less than the continued cost of ensuring permanence for each operation. Operations that are non-permanent but failure-atomic are useful for preserving consistency of objects that can be discarded after failures, but should remain consistent when aborts occur during normal processing.

Underlying any recovery mechanism is an abstract model for failures. Lampson has developed a model that distinguishes between two kinds of failures: errors and disasters [Lampson 81].. Under this model, one of the purposes of recovery is to mask the undesirable properties of real system components by providing new, better-behaved abstract components. These *stable* components function identically to their real counterparts, except that they are not subject to errors. However, stable components remain vulnerable to disasters. By distinguishing between these two kinds of incorrect behavior, the model encourages a clear delineation of the failures that recovery must handle successfully.

For example, reading or writing detectably incorrect data is a storage error, as is *media failure*: the "infrequent" spontaneous decay of correct data. However, reading or writing undetectably corrupted data is a storage disaster. Unlike real storage, *stable storage* always reads and writes data correctly unless a disaster happens. There are several ways to implement stable storage, including duplexed disk or error-correcting RAM with a backup power source.

Incorrect behavior by processors can similarly be classified as erroneous or disastrous. If a processor detects an inconsistency and "crashes" by resetting itself and the system's volatile memory to a standard state, the behavior is considered to be an error. If an inconsistency slips by undetected, then a disaster has taken place. Stable processors that recover from crashes can be built using stable storage to save processor state.

Stable storage gives programmers the ability to make atomic modifications to disk pages or other small, fixed-size units of data. To provide types with failure-atomic or permanent operations, the properties of stable storage must be used to implement atomic modification of arbitrary collections of data. Database systems frequently use *logging* [Gray 78, Gray 81c, Lindsay 79] to achieve failure atomicity and permanence

of transactions. We will briefly summarize this technique and consider its suitability for implementing shared abstract types with these properties.

Unlike "shadow" techniques [Lorie 77, Lampson 81], in which transactions manipulate temporary copies of objects, logging allows transactions to modify objects in place. Furthermore, objects can be transferred between volatile storage (which does not survive processor errors) and non-volatile storage in a way that is independent of transaction commitment. Thirdly, when logging is used, objects themselves do not have to be stored in stable storage. To permit the restoration of consistency if a failure occurs, transactions append information to a *log* in stable storage as they execute. Because objects are modified in place, the following types of inconsistency can be present after a failure:

- Some objects that committed transactions have modified may not have been copied to non-volatile storage prior to the failure. The log must contain sufficient information to redo those modifications during recovery.

- Some objects that incomplete (aborted) transactions have modified may have been copied to non-volatile storage prior to the failure. The log must contain sufficient information to undo those modifications during recovery.

- A media failure may detectably damage the most recent copy of an object on non-volatile storage. The log must have sufficient information to restore the object's current state from an archived version.

Output of the log to stable storage must be coordinated with the commitment of transactions and with the movement of objects between volatile and non-volatile storage. A transaction may not commit until the information needed to redo its modifications has been written to the log. Likewise, a modified object cannot be migrated to non-volatile storage before the information necessary to undo the change has been recorded in the log. This tactic is often referred to as the Write Ahead Log protocol [Gray 78].

There are many ways to represent the required information in the log, but they all have one aspect in common. By definition, a log is a linear sequence of typed records that can only be modified by appending new records at the end. Log records can be read in any order.

Perhaps the simplest way to represent log information is by recording the old and new values of modified objects [Lindsay 79]. Old values can be used to undo aborted transactions; new values can be used to redo committed transactions. The limitations of this representation technique come from its close relation to synchronization policy. If the synchronization rules for an object permit concurrent modification by more than one incomplete transaction, it is frequently impossible to use the old value/new value log representation.

This limitation also applies to recovery techniques based on "shadow" copies.

An abstract type that implements a counter provides a simple example of this limitation. The abstract properties of a counter do not prohibit concurrent increment operations by multiple transactions, as long as the increment operation does not also return the counter's value. Suppose the counter has an initial value of 0. The first increment operation records an old value of 0 in the log. The second transaction records an old value of 1, setting the current value to 2. If the second transaction commits but the first transaction later aborts, restoring the first transaction's old value of 0 is incorrect.

The principles behind this argument can be formalized, and rigorous criteria for the applicability of this log representation can be specified. Our investigation thus far of synchronization for shared abstract types has indicated that there is a lot of concurrency to exploit without violating reasonable type-specific synchronization properties, and synchronization policies that take advantage of this concurrency will not always be compatible with old value/new value logging.

A second logging technique is based on recording transitions rather than old or new states [Gray 81c]. Appropriate inverse transitions can correctly and independently abort forward transitions. For the counter, transition logging records "Increment" for each transaction rather than the counter value. In this case, the inverse operation is to decrement the counter.

The limitations of the transition method come from the difficulty of constructing types with operations that are practical to invert. Sometimes it is difficult to know at the time the log record is written exactly what information will be needed to invert the operation later on. For instance, suppose the counter also offers a reset operation. If a reset occurs and later is aborted, the proper restored value for the counter depends not only on its value at the time of the reset, but also on the operations that have occurred since. Examining the log and redoing these intervening operations may be prohibitively expensive.

The cost and complexity of logging depends on a type's implementation as well as on its abstract properties. For instance, the logging algorithm for a set implemented as a bit vector is quite different from the logging algorithm for a linked-list implementation. Further research is needed to evaluate the power of existing algorithms. This research should lead to new or modified logging techniques that support recovery for a variety of types and implementation strategies.

The composability of types also complicates recovery. In a database, the records at the leaves of the hierarchy are critical. Files and indices serve only to organize this data, and their function is explicitly understood by the system. It is therefore appropriate to provide recovery facilities at the record level; the

system can automatically correct any related file or index structures. In a system allowing general shared abstract types, it is more difficult to decide which operations should be permanent or failure-atomic and which should not. If one type is used in the implementation of another, the recovery behavior of the component type may not be appropriate in the larger context. Like synchronization properties, it is necessary to include recovery properties in the abstract specification for a type.

6. Communication

Communication systems aim to provide useful and efficient communication primitives. Though these goals are easy to state, individual communications systems attempt to meet them in different ways. The communication mechanism of a transaction-based system is used both for the inter-node operation calls that occur within transactions as well as for transaction management operations themselves. The latter group includes transaction initiation, transaction migration, commit coordination, and distributed deadlock detection. Though much is known about communication in transaction-based distributed databases [Lindsay 79, Gray 78], more general transaction-based systems have additional communication requirements and their communication systems must be the subject of more study.

The foremost of these requirements is high communication efficiency. General distributed systems may contain many brief transactions that execute frequently. In current distributed database systems, transactions last at least a few hundred milliseconds, because they perform reads or writes to secondary storage. Performance of the communication system is therefore not critical. General distributed systems, however, will use new types of low-latency stable storage, and very efficient communication is likely to be important. More frequent distributed deadlock detection may also be necessary, especially in real-time systems.

High availability also demands high communication efficiency. For example, frequent operations across node boundaries are required to maintain many data replicas. Communication efficiency can be increased by simplifying protocols, reducing cross-level context switching, and increasing hardware support for the communication system. Communication primitives and their implementations must take advantage of the properties of the underlying communication media and not rely on excessive protocol layering [Spector 82].

For instance, consider remote operation calls on a network. Assume that the network's error rate is low in comparison with the rate of occurrence of other errors such as deadlock. Though remote call primitives could be implemented with complex error-correction facilities, it is only necessary that these primitives have *at-most-once* semantics. That is, the communication system must prevent duplicated, corrupted, or out-of-order operation calls, but it need not guarantee that remote operations are actually executed [Liskov 82b]. If the communication medium is a typical local area network, those semantics can be provided efficiently. It is

deliberately left to the transaction manager to detect and recover from other communication errors (e.g., lost messages) by causing a transaction abort. This is an application of Saltzer's "end-to-end" argument [Saltzer 81].

Other optimizations to transaction communication facilities include batching transmissions and using multicast. Batching can be used to transmit a group of updates that were deferred until commit time. Multicast can be used for the transmission of similar remote operations to multiple sites [Rowe 79], for example, when transactions access replicated data. For sufficiently reliable communications media, multicast messages can be sent without requesting acknowledgments. The error recovery facility of the transaction manager is responsible for recovering from communication errors.

Though eliminating functions from intermediate protocol levels can improve efficiency, there are some problems to consider. For example, flow control and security are often functions of intermediate-level protocols, and, when required, must instead be added to the high-level transaction protocols. Additionally, reflecting many communication errors back to the transaction manager can actually result in lower performance if relatively unreliable communication media are used.

Beyond added communication efficiency, more demanding transaction management operations may induce other new requirements. The transaction coordinator may require that the various nodes participating in a transaction agree to commit their operations *cotemporally*. This problem is relevant in real-time transaction processing where transactions must simultaneously activate several devices.

The cotemporal commit problem is described by Gray as the problem of N generals trying to agree, via exchange of messages along an unreliable path, on a time for simultaneous attack [Gray 78]. Protocols that can be used to solve it are analogous to 2-phase commit protocols but with an added constraint concerning the time the participants actually carry out the commit operation. For a communication medium that can lose messages, there is no protocol that guarantees that the participants will agree to commit cotemporally. However, protocols similar to the centralized 2-phase commit protocols are better than ones similar to the linear commit protocol, because the centralized protocol permits the parallel transmission of messages to the participants. This increased parallelism reduces the interval during which some participants may have agreed to commit at a certain time, whereas others have not yet been so informed.

Increased efficiency and cotemporal transaction commit are only two examples of requirements for communication systems that support general transaction mechanisms. Though such requirements are similar to those of distributed database systems, there are differences that must be studied further.

7. Summary

The goal of constructing a transaction kernel is to make transactions available as a fundamental programming construct for reliable distributed computing, thereby reducing the complexity of designing and implementing reliable distributed systems. There is considerable evidence that transactions free the programmer from continual reimplementation of complex synchronization and recovery code, and that they will be useful in the construction of distributed systems. This paper has suggested the possibility of building a transaction kernel to support transactions containing calls on user-definable shared abstract data types. It has also described important research questions, such as what modifications to the traditional transactional model will be necessary, and whether systems built using transactions will have acceptable efficiency.

We are attempting to answer these questions at Carnegie-Mellon, in an effort that overlaps with the Archons project and currently includes five researchers. Specifically, we are pursuing research on the topics indicated by the major section headings of this paper:

- Extensions to the transaction model and the overall structuring of distributed systems that utilize transactions, including the identification of useful shared abstract data types.
- The specification and lock-based implementation of synchronization for shared abstract types.
- The impact of high-concurrency shared abstract types on deadlock detection and resolution algorithms.
- The specification and implementation of recovery for shared abstract types.
- The fulfillment of communication requirements for systems utilizing transactions.

There are other issues concerning the general use of transactions, but this subset forms a good basis for research on extending their utility. We are not considering deadlock avoidance mechanisms, alternatives to lock-based synchronization, or incorporation of transactions into programming languages. Work that overlaps ours and also addresses some of these other topics is occurring elsewhere [Liskov 82b, Allchin 82]. When the results of present research on transactions become available, it should be possible to construct a transaction kernel that encourages more universal use of transaction-based programming.

Acknowledgments

We gratefully acknowledge the contributions of Dean Daniels and Cynthia Hibbard, who helped us organize and present the ideas in this paper.

References

[Allchin 82] James E. Allchin and Martin S. McKendry.
Object-Based Synchronization and Recovery.
1982.
School of Information and Computer Science, Georgia Institute of Technology, submitted for publication.

[Bernstein 81] P. A. Bernstein, N. Goodman, and M. Y. Lai.
Two Part Proof Schema for Database Concurrency Control.
In *Proc. Fifth Berkeley Wkshp. on Dist. Data Mgmt. and Computer Networks*, pages 71-84. February, 1981.

[Daniels 82] Dean Daniels.
Query Compilation in a Distributed Database System.
Master's thesis, MIT, March, 1982.

[Digital Equipment Corporation 72]
Decsystem10 Assembly Language Handbook
2 edition, Digital Equipment Corporation, Maynard, MA, 1972.

[Eswaran 76] K. P. Eswaran, James N. Gray, Raymond A. Lorie, I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Comm. of the ACM 19(11), November, 1976.

[Garcia-Molina 82]
H. Garcia-Molina.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
Technical Report 285, Department of Electrical Engineering and Computer Science, Princeton, June, 1982.

[Gifford 79] David K. Gifford.
Violet, an Experimental Decentralized System.
In *Proceedings I.R.I.A. Workshop on Integrated Office Systems.* Versailles, France, November, 1979.
Also available as Xerox Palo Alto Research Center Report CSL-79-12.

[Gray 75] J. N. Gray, R.A. Lorie, G. R. Putzolu, and I. L. Traiger.
Granularity of Locks and Degrees of Consistency in a Shared Data Base.
IBM Research Report RJ1654, IBM Research Laboratory, San Jose, Ca., September, 1975.

[Gray 78] James N. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, and G. Seegmuller (editors), *Lecture Notes in Computer Science*. Volume 60: *Operating Systems - An Advanced Course*, pages 393-481. Springer-Verlag, 1978.
Also available as IBM Research Report RJ2188, IBM San Jose Research Laboratories, 1978.

[Gray 80] Jim Gray.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, Ca., August, 1980.

[Gray 81a] Jim Gray.
The Transaction Concept: Virtues and Limitations.
In *Proc. of Very Large Database Conference*, pages 144-154. September, 1981.

[Gray 81b] Jim Gray, Pete Homan, Ron Obermarck, Hank Korth.
A Straw Man Analysis of Probability of Waiting and Deadlock.
IBM Research Report RJ3066, IBM Research Laboratory, San Jose, Ca., February, 1981.

[Gray 81c] James N. Gray, et al.
The Recovery Manager of a Data Management System.
ACM Computing Surveys (2):223-242, June, 81.

[Hoare 74] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Comm. of the ACM 17(10):549-557, October, 1974.

[Jacobson 82] David M. Jacobson.
Transactions on Objects of Arbitrary Type.
Technical Report 82-05-02, University of Washington, May, 1982.

[Korth 81] Henry F. Korth.
A Deadlock-Free Variable Granularity Locking Protocol.
In *Proc. Fifth Berkely Wkshp. on Dist. Data Mgmt. and Computer Networks.* February, 1981.

[Korth 83] Henry F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1), Jaunary, 1983.

[Kung 79] H. T. Kung and C. H. Papadimitriou.
An Optimality Theory of Concurrency Control for Databases.
In *Proceedings of the 1979 SIGMOD Conference.* ACM, Boston, MA., May, 1979.

[Lampson 81] Butler W. Lampson.
Atomic Transactions.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science*. Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 11pages 246-265. Springer-Verlag, 1981.

[Lin 82] Wen-Te K. Lin, Jerry Nolte.
Performance of Two Phase Locking.
In *Proceedings 7th Berkeley Conference on Distributed Data Management and Computer Networks*. February, 1982.

[Lindsay 79] Bruce G. Lindsay, et al.
Notes on Distributed Databases.
IBM Research Report RJ2571, IBM Research Laboratory, San Jose, Ca., July, 1979.

[Liskov 82a] Barbara Liskov and Robert Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In *Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 7-19. Albuquerque, NM, January, 1982.

[Liskov 82b] Barbara Liskov.
On Linguistic Support for Distributed Programs.
IEEE Trans. on Software Engineering SE-8(3):203-210, May, 1982.

[Lomet 77] David B. Lomet.
Process Structuring, Synchronization, and Recovery Using Atomic Actions.
ACM SIGPLAN Notices 12(3), March, 1977.

[Lorie 77] Raymond A. Lorie.
Physical Integrity in a Large Segmented Database.
ACM Trans. on Database Systems 2(1):91-104, March, 1977.

[Moss 81] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, MIT, April, 1981.

[Reed 78] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, MIT, September, 1978.

[Rosenkrantz 78] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis.
System Level Concurrency Control for Distributed Databases.
ACM Trans. on Database Systems 3(2), June, 1978.

[Rowe 79] Lawrence A. Rowe, Kenneth P. Birman.
Network Support for a Distributed Data Base System.
In *Proceedings 4th Berkeley Conference on Distributed Data Management and Computer Networks*, pages 337-352. August, 1979.

[Saltzer 81] J. H. Saltzer, D.P. Reed, D.D. Clark.
End-To-End Arguments in System Design.
In *Proceedings 2nd International Conference on Operating Systems*, pages 519-512. Paris, France, April, 1981.

[Schwarz 82] Peter M. Schwarz, Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Carnegie-Mellon Report CMU-CS-82-128, Carnegie-Mellon University, Pittsburgh, PA, September, 1982.

[Sha 83] Lui Sha, E. Douglas Jensen, Richard F. Rashid, J. Duane Northcutt.
Distributed Co-operating Processes and Transactions.
In *Proceedings ACM SIGCOMM Symposium.* 1983.

[Spector 82] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Comm. of the ACM 25(4), April, 1982.

[Weihl 81] William E. Weihl.
Atomic Actions and Data Abstractions.
1981.
MIT Laboratory for Computer Science.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DTIC

8-86